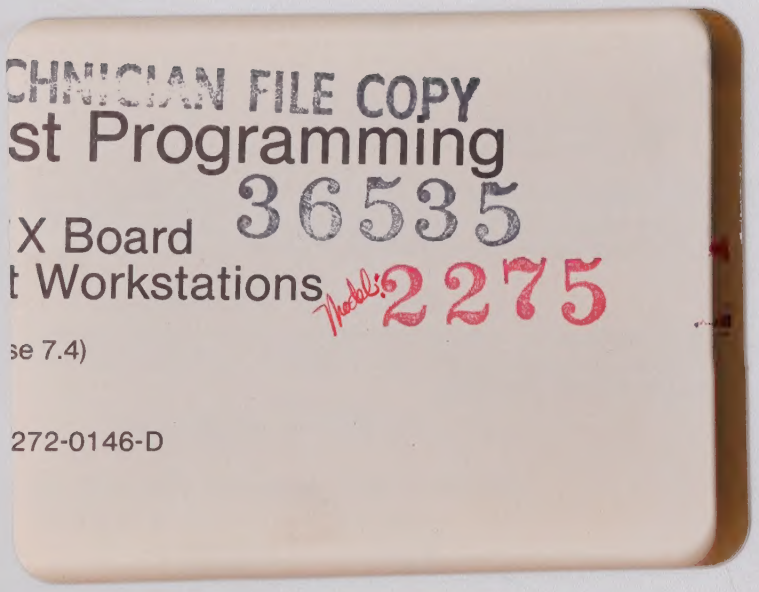
GenRad



Instruction Manual

Contents

Introduction – Section 1
Statement of Syntax – Section 2
Types of Statements – Section 3
Testing the UUT – Section 4
Reacting to a Test Failure – Section 5
Test Language Descriptions – Section 6
Appendices
Index

TECHNICIAN FILE COPY

Test Programming

227X Board Test Workstations

36535
Tektronix 2275

(Release 7.4)

Form 2272-0146-D

©GenRad 1983
Concord, Massachusetts, U.S.A. 01742
November, 1983

List of Effective Pages

INSERT LATEST CHANGED PAGES. DESTROY SUPERSEDED PAGES.

NOTE: The portion of the text affected by the changes is indicated by a vertical line in the outer margin of the page.

TOTAL NUMBER OF PAGES IN THIS PUBLICATION IS 286 **CONSISTING OF THE FOLLOWING:**

Issue Original **Date** November, 1983

Page No.	Issue
Title	November, 1983
A	November, 1983
iii through x	Original
1-1 through x	Original
2-1 through 2-5	Original
3-1 through 3-17	Original
4-1 through 4-59	Original
5-1 through 5-7	Original
6-1 through 6-150	Original
A-1 through A-4	Original
B-1 through B-3	Original
C-1 through C-13	Original
D-1 through D-6	Original
I-1 through I-3	Original

NOTE

The software package, described in this manual and supplied by GenRad, Inc. as a component of a system or as a separate item, is furnished for use on a single CPU. All proprietary rights are reserved by GenRad, Inc.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Computer Software clause in DAR 7-104.9(a).

GenRad Inc., Concord, MA

The following are trademarks of GenRad, Inc.:

BUSBUST	SCRATCHPROBING
GRnet	SpeedPlus
NIM	TRACS

NOTE

The material in this manual is for informational purposes only and is subject to change without notice. GenRad assumes no responsibility for any errors or for consequential damages that may result from the use or misinterpretation of any of the procedures in this publication.

Table of Contents

Section and Paragraph — Page

FOREWORD
General v
Hardware Configuration vi
Test Programming Statement Differences ix
Operational Verification Commands ix
Optional UUT Power Supplies x

1 INTRODUCTION
1.1 General 1-1
1.2 Scope of Manual 1-1
1.3 Related Manuals 1-2

2 STATEMENT SYNTAX
2.1 General 2-1
2.2 Comments 2-1
2.3 Command Keyword 2-2
2.4 Terminator 2-3
2.5 Labels 2-3
2.6 Statements 2-4
2.7 Variables 2-4
2.8 Operators 2-4
2.9 Scaling Factors 2-5

3 TYPES OF STATEMENTS
3.1 Organization 3-1
3.2 Initializing the Test Program 3-3
3.3 Analog Instrumentation Statements 3-4
3.4 Auxiliary Test Statements 3-4
3.5 Flow Control Statements (Non-Burst) 3-6
3.6 Data Manipulation Statements 3-9
3.7 Input/Output Statements 3-11
3.8 Initializing the Logic 3-12
3.9 Initiate a Burst 3-12
3.10 Digital Testing 3-13
3.11 Flow Control in a Burst 3-13

Section and Paragraph — Page

3.12 Terminate a Burst 3-16
3.13 Terminate the Program 3-16
3.14 Subroutines (Non-Burst) 3-16

4 TESTING THE UUT
4.1 Hardware and Software Devices 4-1
4.2 Analog Test Devices 4-4
4.3 Digital Test Devices 4-27
4.4 SpeedPlus Test Devices 4-34
4.5 External Test Devices 4-49
4.6 UUT Power Supplies 4-51
4.7 Utility Devices 4-54
4.8 Instrumentation Statement Format 4-56

5 REACTING TO A TEST FAILURE
5.1 General 5-1
5.2 Fail Vector 5-1
5.3 User Failure-Reporting Options 5-3
5.4 Error Commands 5-7
5.5 Burst Errors 5-7

6 TEST LANGUAGE DESCRIPTIONS
6.1 General 6-1
6.2 General Utility Statements 6-2
6.3 Analog Instrumentation Statements 6-43
6.4 Digital Testing 6-93

APPENDIX A — GLOSSARY OF TERMS

APPENDIX B — INSTRUCTION SUMMARY

APPENDIX C — PREPARING EXTERNAL SUBROUTINES

APPENDIX D — SET AND MEAS ON THE I488 BUS

INDEX

Digitized by the Internet Archive
in 2024

https://archive.org/details/manualsplus_02477

Foreword

GENERAL

This manual describes test programming on the 2270, 2271, 2272, and 2275 Board Test Workstations. The same programming language is employed on all 227X systems. However, because these 4 systems differ somewhat in hardware configuration, test programming varies slightly among them. Specific differences are covered in the appropriate sections of this manual. General differences and important considerations are outlined below.

Any circuit board test can be programmed by creating a test description in the 227X test language described in this manual. But because 227X test systems are capable of writing test programs automatically from circuit description data, many users will not need to become proficient in this test language. Any user, however, who needs to understand or modify the tests written by the 227X Automatic Test Generation facility will require some acquaintance with the test language. Users who must write their own test programs or who must add model tests to the analog or digital test libraries will need more thorough understanding of the testing principles and language statements described here. The library programming languages described in the Test Library Programming manual are based on this test language.

The 2271, 2275 and 2272 have multiplexed driver/sensor systems: The 2271 and 2275 share 4 driver/sensors for every 16 nails, and the 2272 uses 2 driver/sensors for each set of 16 nails. In every test program test nails must be assigned to all tested nodes on the unit-under-test (UUT); however, on the multiplexed systems nails must be assigned to nodes in such a way that multiplexing conflicts do not arise. In each 16-nail group, only 4 (2271 and 2275) or 2 (2272) test nails can be used at one time for UUT testing. To relieve the test programmer (or ATG) of the task of finding nail assignments that do not cause multiplexing conflicts, a special program in the system software, called nail assignment, reassigns the nodes used in a test program to new nail numbers, to permit conflict-free multiplexing. It is strongly urged that, to prevent possible conflicts in digital testing, nail assignment software be run as part of all test program development, changes, or additions on the 2271, 2272 and 2275.

In test development, ATG is first run using temporary nail names (often the UUT node names) to create an error-free test program. Nail assignment then processes this test program to select an actual test nail for each node within constraints of the available system hardware. Hence, a test program prepared on one 227X system can usually be remapped to another 227X system, if that system has the necessary hardware available. In addition, nail assignment generates wire lists to direct construction of the test fixture and changes the temporary nail names in the test program to the assigned ones.

HARDWARE CONFIGURATIONS

Figure 1 shows the basic elements of the 2270 Workstation, which can have several possible combinations of analog, digital, and hybrid (usable for analog or digital) nails. Each digital or hybrid nail has one dedicated driver/sensor assigned to it, and each analog or hybrid nail can be connected to any of the 4 instrument buses A through D.

Figure 2 shows the 2271 and 2275 Workstations, in which every test nail is hybrid (can perform digital and analog testing). In these systems, each group of 16 nails can simultaneously access up to 4 lines from among the 4 analog instrument buses and/or 4 driver/sensors. Each scanner driver/sensor card in the 2271 contains 4 driver/sensors muliplexed to 16 test nails; in the 2275 each card contains 2 such 4X16 matrices. Nail assignment software is used in test fixture and program preparation to preclude possible conflicts in digital testing.

Figure 3 shows the configuration of the 2272 test system. It is similar to a totally hybrid 2270 with the addition of a pin expander (PEX) board for every driver/sensor board and scanner board. The PEX board expands each group of 16 hybrid lines onto 128 nails, with 2 driver/sensors available for every 16 nails. Nail assignment software is used to help prepare test fixture wiring and the test program.

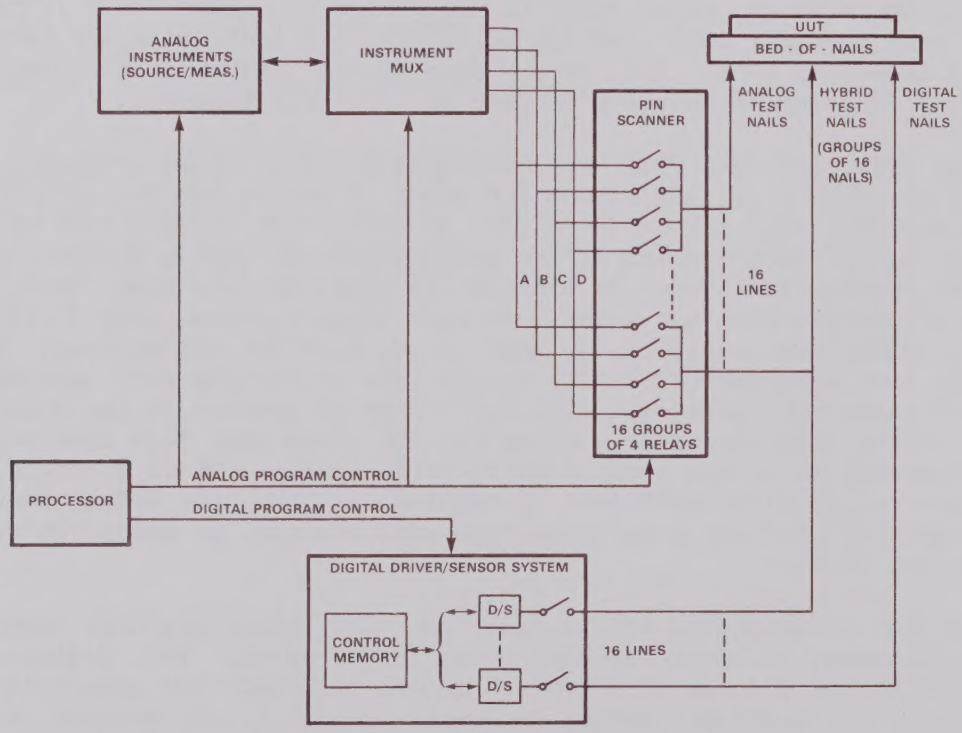


Figure 1. Block Diagram of the 2270 Board Test Workstation.

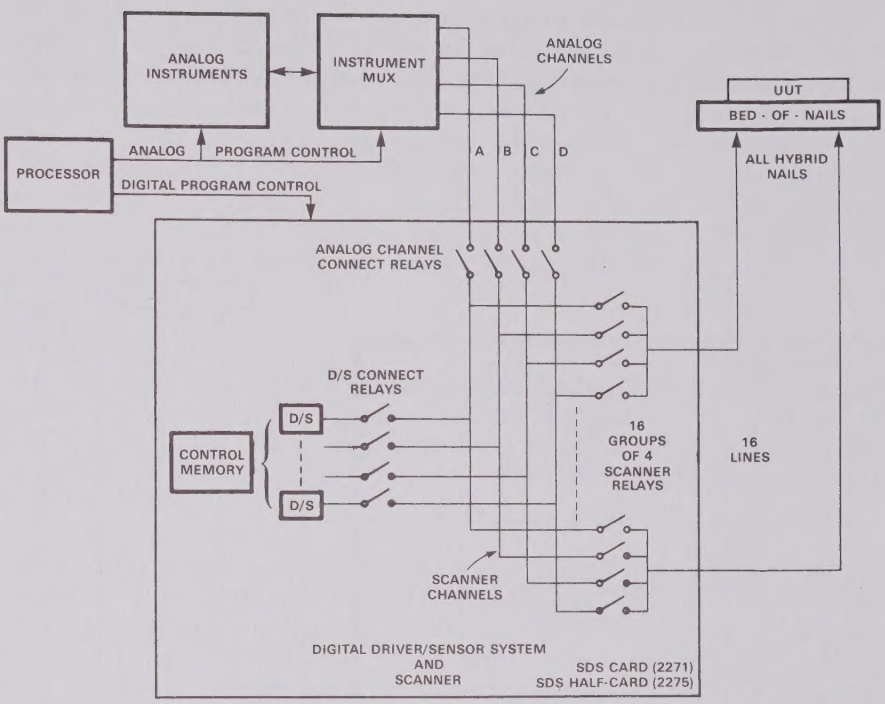


Figure 2. Block Diagram of 2271 and 2275 Board Test Workstations.

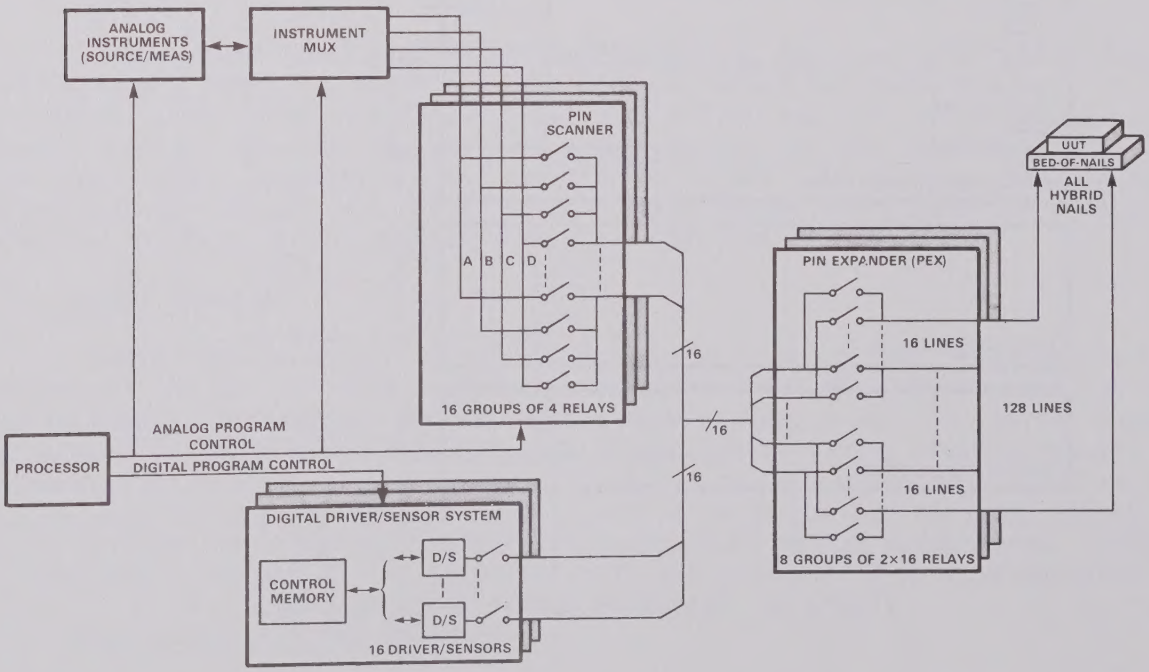


Figure 3. Block Diagram of the 2272 Board Test Workstation.

Both the 2271 and 2272 systems have available an optional SpeedPlus feature. This option provides high-speed driver/sensors ("fast" test nails), and special timing nails for UUT clock generation/synchronization and for triggering tests on unique UUT events. A block diagram of the SpeedPlus nail types is shown in Figure 4. A detailed description of this feature can be found in Section 4.

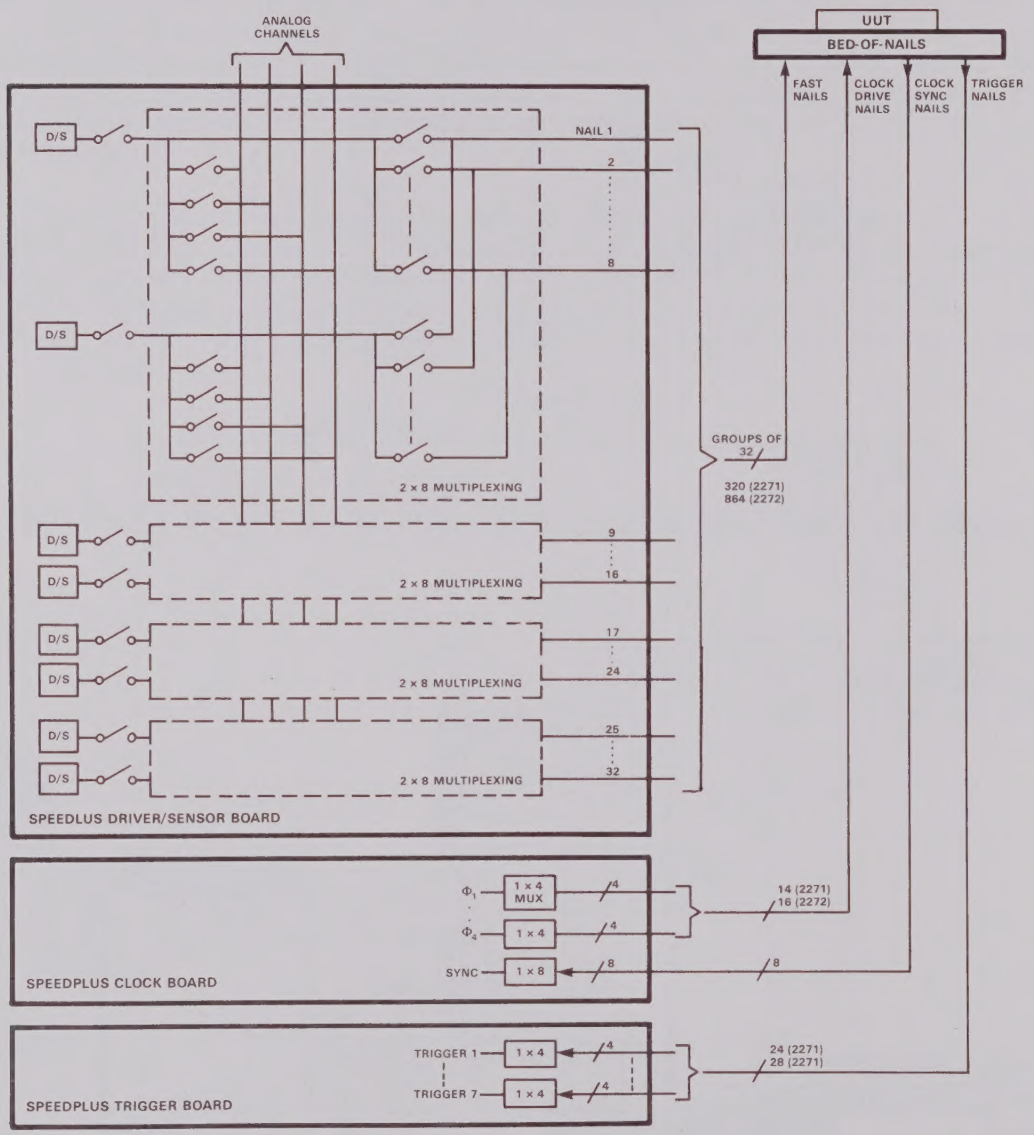


Figure 4. SpeedPlus Option Block Diagram.

TEST PROGRAMMING STATEMENT DIFFERENCES

SHORTS and OPENS Testing

SHORTS and OPENS testing on the 2270 provide analog tests for analog and hybrid nails but only digitally driven tests for digital nails. Because the 2271, 2275, and 2272 are all hybrid and each nail has access to the system analog instrumentation, SHORTS and OPENS testing on these machines are completely analog. This also eliminates the need in these systems for grouping the types of nails used in SHORTS and OPENS testing. The OPENS and SHORTS command descriptions in para 6.3 cover these differences.

SCAN Statement

Due to nail multiplexing considerations, the SCAN keywords CLOSE, OPEN, MBB (make before break), BBM (break before make), # (all but), and the bit string variable are not available for use in 2272 test programming. Only the default keyword JAM is recognized in a 2272 SCAN statement. Refer to para 6.3 for more information.

Most effects of multiplexing are transparent to the user, unless it becomes necessary to access a specific physical nail. The temporary nail numbers used before nail assignment will often differ completely from the numbers provided by nail assignment for later use. When using the SCAN statement in multiplexed systems, it is important to maintain consistency between the specified nails and the numbering scheme appropriate to the test program using the SCAN statement.

OPERATIONAL VERIFICATION COMMANDS

General

For sophisticated test system users, such as those writing operational verification procedures, there are additional commands for individually controlling 2271, 2272, and 2275 multiplexing relays and the driver/sensor current sources. Similar commands are also provided for the SpeedPlus subsystem. These commands are not required in normal test programs because the system software manages all multiplexing. Their misuse can damage the system hardware, and they should therefore be used with caution.

SDSOPT Command

SDSOPT provides the user direct control over the driver/sensor current sources and the 2271 or 2275 analog channel connect relays. This command can be used to measure performance of test system driver/sensors and UUT digital inputs and outputs in these systems by allowing connection of analog instrumentation to active driver/sensors. It also allows activation of the driver/sensor current sources used for the Scratchprobing diagnostic technique and for 2270 digital shorts testing, enabling performance verification by analog measurement. SDSOPT is described in detail in the 2270/2271/2275 and 2272 system Service manuals.

SCNCLOSE and SCNOPEN Commands

These commands allow user control of the analog scanner relays in the 2272 test system. This is useful in writing operational verification procedures or measuring performance of digital circuitry. The 2272 Service manual describes these commands.

PEXCLOSE, PEXGND, and PEXOPEN COMMANDS

These commands directly control relays in the 2272 Pin Expander (PEX) boards and are used for special OVP programming. Refer to the 2272 Service manual for more information.

SpeedPlus Relay Commands

Special commands are also available for directly controlling the relays in the optional SpeedPlus subsystem. For descriptions of these commands (listed in Table 2-1), refer to the Service manual for your system.

OPTIONAL UUT POWER SUPPLIES

While the 2270 can contain one High Current Supply (HCS) for UUT power, the 2271 can have up to 2 and the 2272 up to 4. The HCSs are separately programmed for voltage output, current limit, and relay connection to the test head. HCS programming is covered in para 4.6 and 6.3 of this manual.

Additionally, the test head receiver pinout for the various power supplies differs among the 2270, 2271, and 2272. Paragraph 4.6 contains information on connecting and programming all UUT power supplies.

Introduction – Section 1

1.1 GENERAL

Automatic testing of pc boards on the 227X system is controlled by test programs written in the 227X system test language. Typically these programs are written by the system's Automatic Test Generation (ATG) facility, using model tests for the devices on the board that ATG finds in the system's test libraries. A test program is then usually checked and modified by the user. The program consists of a sequence of test language statements that define the order and type of measurements to be performed, manipulate and analyze the data, and communicate with the user on the progress and results of the test.

The test language is high-level and therefore similar to English sentence structure. For example, the test language statement:

```
IF VTEST<10 THEN WRITE 'TEST FAILED';
```

reads as follows: If the voltage value stored in VTEST is less than 10 V, then display the message TEST FAILED.

The instrumentation statements, used to control the system test devices, are task-oriented and reflect the electrical attributes of each device. For example,

```
SET DCS V=5;
```

sets up the dc source to supply a voltage of 5 V and

```
MEAS Z INTO ZVAL AT(15:23);
```

directs the ac measurement unit to measure the impedance between pins 15 and 23 of the unit-under-test (UUT) and store the measured value in ZVAL.

Test programs for in-circuit testing of analog and digital components are generally produced by the Automatic Test Generation (ATG) feature. The final output from ATG is a source test program written in the test language described in this manual. However, users may develop their own test programs without the aid of ATG or they may take ATG-produced programs and modify them as desired. Source programs written in this test language are then converted to machine-coded object programs by the system translator. Refer to the 227X Test Set Preparation manual for a description of ATG and TRANSLATE modes.

1.2 SCOPE OF MANUAL

The purpose of this manual is to describe the 227X test language so that a user can either create a new test program or modify an existing one. Previous programming experience is not a prerequisite. This manual is structured to accommodate different users who may have varying degrees of programming experience. Some sections are designed especially to introduce a new user to the test system and the test language. Other sections are aimed at providing quick reference information for the more experienced user. Sections 4 through 7

and Appendix A are tutorial in nature and thus fit into the first category. Section 6, which contains the bulk of the detailed test language descriptions, is more of a reference section. It defines the format and presents examples, rules and considerations for each statement and test device in the system.

The manual is organized into the following sections:

Section 1 - INTRODUCTION introduces the reader to the manual.

Section 2 - LANGUAGE SYNTAX illustrates the format of the test language statements and presents syntactical rules and considerations applicable to writing test language statements.

Section 3 - TYPES OF STATEMENTS introduces the user to all 227X test language statements and groups them into categories that fit into a typical test program structure.

Section 4 - SYSTEM TEST DEVICES introduces the user to all test devices available in this system, how they are used and how they are controlled by the test language. This section also describes the format and parameters allowed in all instrumentation statements.

Section 5 - REACTING TO TEST FAILURES describes all the options available to a user when a UUT failure is encountered.

Section 6 - TEST LANGUAGE STATEMENTS presents in alphabetical order, for easy reference, a detailed description of each statement and test device; all non-digital statements/devices are covered first, followed by all statements/devices reserved for digital testing only.

Appendix A - GLOSSARY OF TERMS describes some of the terms associated with the 227X test language and referred to throughout the manual.

Appendix B - INSTRUCTION SUMMARY lists and summarizes the function of each test device and each test language statement.

Appendix C - EXTERNAL SUBROUTINES describes the procedure for developing and incorporating into the system external subroutines written in high-level languages other than the 227X test language.

NOTE

This manual covers all available 227X test language commands. The actual execution of some of these commands, however, depends on the test hardware configuration of the system being used. For example, digital testing can only be performed by systems containing driver/sensors. ATG and the test language translator operate independently of the actual test hardware and thus allow a user to compile any 227X program on any 227X system regardless of system configuration. At execution time, the run-time system determines if the necessary test hardware is available and issues an error message if it is not.

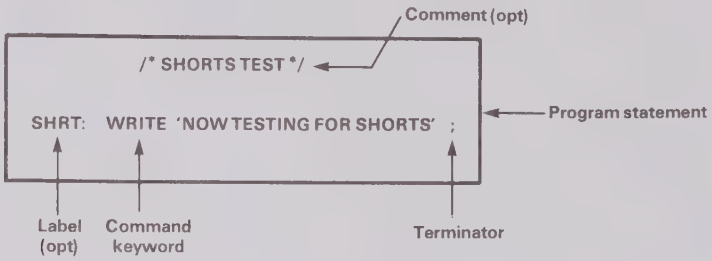
1.3 RELATED MANUALS

The complete set of GenRad manuals furnished with each circuit board test system is described in the Document Directory. Refer to that manual for part numbers and ordering information.

Statement Syntax – Section 2

2.1 GENERAL

As shown in the following illustration, a typical 227X test language statement consists of the following components: Comment (optional), label (optional), command keyword (and other parameters), and a terminator.



2.2 COMMENTS

Comments are typically used to describe the function of individual statements or program segments, in order to make the program listing easier to understand. Comments can be included before or after any statement, and must be bracketed between a /* and */ character sequence. The translator ignores all characters in a comment string. As a result, comments have no effect on the execution of a test program.

A line may have multiple comments, or a comment may occupy more than one line. For example,

```
/* THIS COMMENT BEGINS AT THE START OF THIS LINE AND ENDS DOWN ON THIS LINE -
RIGHT HERE.*/

/* THIS LINE HAS*/ /*THREE*/ /*COMMENTS*/
```

Also, comments can be nested, i.e., smaller comments can be contained within a larger comment string.

```
/* TYPICAL POWER SKELETON
SET PIO(0).............;/*ATTACH UPS TO PCB*/
SET DCV UPS(4).........;/*TURN ON UPS*/
*/
```

In this example, each statement has its own comment, and the 2 statements together form a larger comment string. As a result, the 2 statements are not executed by the program.

2.3 COMMAND KEYWORD

Each statement must begin with a command keyword (Table 2-1) that designates the operation being performed (e.g., WRITE, TEST, DELAY). Following this command keyword may be additional entries indicating function, device, and other parameters applicable to the type of statements specified.

Table 2-1
TEST LANGUAGE KEYWORDS

ACTIVE	DCV	FSCNCLOSE*	LVLB	PEXOPEN**	T*
ACZ	DECLARE	FSUB	MAX	PHASELOCK*	TEST
ANA	DELAY	FTYPE	MAXDLY	PIO	TH*
AND	DELTA	GLOBAL	MAXTIME	PMSG	THEN
ARITH	DEV	GOSUB	MBB	PN	TIME
ASSIGN	DEVTYPE	GOTO	MEAS	PU	TINC*
AT	DIG	GROUP	MESFILE	Q	TL*
BBM	DLY	HFORCE	MODE	R	TO
BIAS	DMSG	HI	MUX	RATIO	TOL
BITCLR	DO	HIPREC	NAILOFFSET	RDLY	TPS
BITSET	DST*	HRLY	NAME	READ	TRGGND*
BRANCH	ED*	I	NAND	RECOMPILE	UNLOAD
BREAK	ELSE	I488	NINC*	REF	UPS
BSTRING	END	IC	NOFSUB	RETURN	USE
BURST	EXIT	ID	NOPHASELOCK*	RLY	USING*
BUS	EXITIF	IF	NOPRINT	RLYCD*	V
BUSTEST	EXTERNAL	IG	NOPROBE	RLYCDEX*	VA
CALL	F	IH	NOR	RLYCS*	VAL
CDA*	FAIL	II	NOT	RLYCSEX*	VB
CDB*	FAILS	IL	OFF	RLYT*	VCN
CDC*	FAST*	INDICT	OG	RLYTEX*	VCDH*
CDD*	FCMSG	INTO	OH	RLYTEXT*	VCDL*
CHA	FDIG	IS	OI	RM	VCST*
CHB	FDMSG	IST	OL	RP	VIHA
CHC	FEXOPEN*	ITT	OM	RS	VIHB
CHD	FEXCLOSE*	ITTL	ON*	S*	VILA
CLEAR	FEXGND*	JAM	ONABORT	SCAN	VILB
CLKGND*	FHI	KEYBD	OPEN	SCANOFFSET	VL
CLOCK*	FILE	LED	OPENS	SCNCLOSE**	VMAX
CLOSE	FLABEL	LENGTH	OR	SCNOPEN**	VOHA
CMD	FLAG	LET	OS	SDIG	VOHB
CMSG	FLAGFAIL	LFORCE	OTTL	SDSOPT	VOLA
CP	FLO	LO	OU	SET	VOLB
CS	FLOOP*	LOAD	OUT	SHORTS	VTT*
CSTRING	FLOAT	LOGFILE	PASSES	SIGNAL*	WHILE
CSYNC*	FMEAS	LOOP	PAUSE	STEPS	WRITE
D	FNAILOFFSET*	LOOPIF	PB	STM	XOR
DATE	FORCE	LP	PERIOD*	SUBROUTINE	XP
DCI	FPMSG	LS	PEXCLOSE**	SUBSTR	XS
DCM	FSTAT	LVLA	PEXGND**	SWITCH	Z
DCS	FSCNOPEN*				

*SpeedPlus option only
**2272 only

For example, in the following statement:

```
TEST CP F=10K VAL=470P TOL=10 AT(10:11:12:12);
```

TEST is the command keyword, CP is the function (parallel capacitance), and F, VAL, TOL, AT are specific parameters for that statement (i.e., Frequency=10KHz, VALue=470 pF, TOLerance=10%, AT( )= scanner pin connections).

The digital TEST LGC and SET LGC statements are exceptions in that they do not require an explicit command keyword. TEST LGC is the default statement in the test language and thus needs no command keyword. That is,

```
TEST LGC OH(22) OL(23);
```

can be written

```
OH(22) OL(23);
```

Similarly, for convenience in writing digital tests, the command keywords SET LGC can be replaced with the symbol ($). For example,

```
$ IH(2-5);
```

is equivalent to

```
SET LGC IH(2-5);
```

2.4 TERMINATOR

Every test language statement must terminate with a semicolon (;).

2.5 LABELS

A label is an optional identifier up to 11 characters long, used as a statement address. Several rules and considerations are associated with the use of labels.

1. Labels may contain alphanumeric characters and the underscore (_) symbol.

2. The test language keywords listed in Table 2-1 cannot be used as labels.

3. Labels must be terminated by a colon (:) when preceding a statement, but cannot have a colon when referenced within a statement. For example,

```
LABEL1:  IF A=B THEN BRANCH LABEL2 ;
   ^                          ^
   |                          |
colon                    no colon
```

4. Whenever a branch construct is used, the target (destination) of the branch must be identified by a label. For example

```
        BRANCH TEST2;
            |
            |
TEST2:  TEST R . . . ;
```

5. Statements may have more than one label. For example,

```
L1: L2: L3: PAUSE;.
```

6. Purely numeric labels are allowed (e.g., 1234:).

7. A comment string is not allowed between the body of a label and the colon (e.g., LABEL /*comment-string*/: is not allowed).

8. Labels are also used in Run-Time System error messages to identify the error location. Refer to the 227X Test Set Preparation manual.

2.6 STATEMENTS

Test language statements may be either simple or compound. A simple statement is a single, stand-alone statement with command and parameter specifications, e.g.,

```
WRITE 'TEST COMPLETE';
```

A compound statement is a sequence of zero or more statements delimited by brackets ([ ]) and terminated by semicolons (;), as shown in the examples below. The syntax of a compound statement is identical to that of a simple statement, so that compound statements can be used anywhere that simple statements are used. Typically, compound statements are used in flow-control constructs or as [error-command] specifications in instrumentation statements.

Examples

1.
```
IF VVAL>5 THEN
[
WRITE 'TEST FAILED';
BRANCH NEXT;
];
```

2.
```
TEST ARITH RVAL LO=0 HI=10
[
WRITE 'OUT OF RANGE';
BRANCH NEXTTEST;
];
```

2.7 VARIABLES

All quantities subject to change by the program must be identified as variables via DECLARE statements at the beginning of the program. Variable names are user-assigned and can be up to 10 characters long. The DECLARE statement defines the variable type (floating-point, character-string, bit-string or file) and the scope of the variable (local, global or load).

The format or value that a variable can assume is limited to the type specified in the DECLARE statement (e.g., a character-string variable cannot take on a floating-point value). The scope of the variable indicates its effective life-span. Local variables exist only in the program block (i.e., main body or subroutine) in which they are DECLAREd. Global variables exist throughout the program. Local and global variables are re-initialized by the system at the start of each program execution. Load variables exist and retain their values throughout all program executions, from program load to program unload time.

2.8 OPERATORS

Operators are special characters that have some significant meaning in the test language. There are several types of operators:

Arithmetic (Shown in order of precedence)

Operator	Meaning
*	multiply
/	divide
+	add
-	subtract

Relational (Comparison of 2 values)

>	more positive than (less negative than)
>=	more positive than (less negative than) or equal to
<	more negative than (less positive than)
<=	more negative than (less positive than) or equal to
=	equal to
<>	not equal to

Character Strings

!!	concatenate 2 character strings (i.e., link 2 character strings together)

2.9 SCALING FACTORS

To simplify the specification of numerical quantities within program statements, letter suffixes representing powers of 10 are allowed, as shown in the following table:

MEG	$= X10^{6}$	U	$= X10^{-6}$
K	$= X10^{3}$	N	$= X10^{-9}$
M	$= X10^{-3}$	P	$= X10^{-12}$

E Base 10 exponential form. The letter E is followed immediately by the exponent (preceded by a minus sign (-) if a negative exponent is being specified).

Examples:

52E-4	(=0.0052)
52E4	(=520,000)
I=7.5M	(=7.5 milliamps)
C=.01U	(=.01 microfarad)

Types of Statements – Section 3

3.1 ORGANIZATION

The general organization of a typical 227X test program is shown in Figure 3-1. The test programs written by ATG generally follow this format.

First, the program is initialized by identifying (DECLAREing) to the translator all user-defined variables and subroutines. Then, all non-digital testing is performed, starting with a continuity check and followed by the in-circuit testing of all analog components. In addition to the statements that control the test devices, the 227X test language also contains statements that indirectly aid the test program. These include flow control, input/output, data manipulation statements, and instructions to the operator.

When analog testing is complete, if the UUT contains digital logic and the system has digital or hybrid test pins, power and ground are applied to the UUT, logic levels are defined, and the digital ICs are tested. Unlike analog tests, digital tests are not conducted under immediate control of the test system's computer. Instead, they are down-loaded to a special controller, and the test statements are executed in a high-speed burst mode. Hence, the portion of a test program containing digital test statements is called a burst. Normally each IC is tested within a separate burst. Because digital testing is conducted in a fundamentally different way from analog testing, it uses a different set of test commands and flow control statements. For example, a jump to a different part of the program is performed with a BRANCH statement in the analog portion of a test, but within a burst, a GOTO statement is used.

On systems having the SpeedPlus option, a third set of statements is available to control special functions of the SpeedPlus hardware. Some of these statements appear in the analog portion of a test, and some are used in bursts.

When digital testing is complete, the main body of the program is terminated and all general (non-burst) subroutines are defined.

To introduce the user to all the test language statements, in this section the statements are grouped into the categories shown by the typical test program structure in Figure 3-1.

1. Initialize test program (DECLARE)

2. Analog instrumentation statements (SET, TEST, MEAS, CLEAR device)

3. Auxiliary test statements (DELAY, PAUSE, NAILOFFSET, USE, INDICT)

4. Analog Flow Control:

   Branching (BRANCH, IF-THEN-ELSE)
   Looping (WHILE DO)
   Subroutines (CALL)
   Exiting (EXIT)

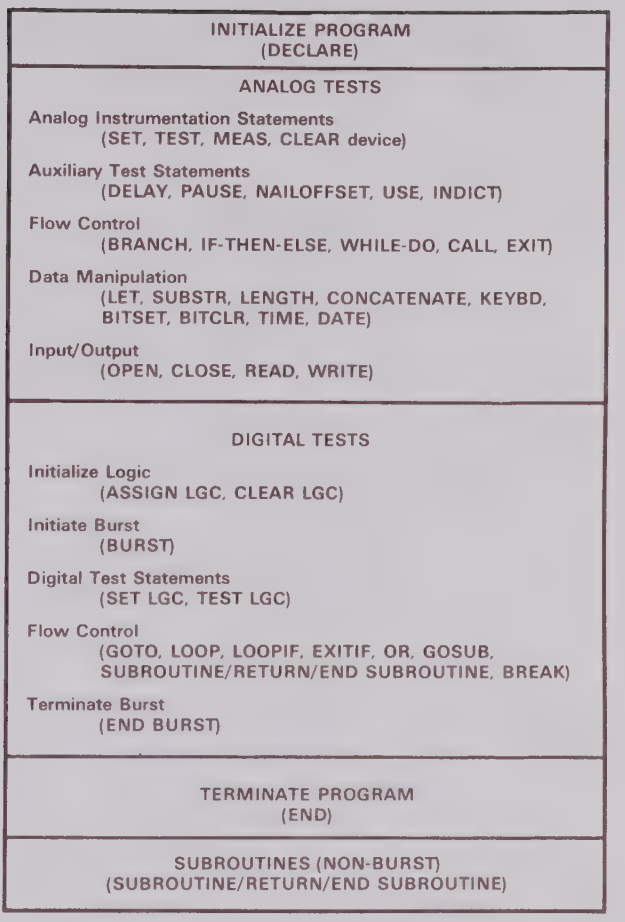


Figure 3-1. Typical Test Program Structure.

5. Data Manipulation:

   Variables (LET)
   Character Strings (SUBSTR, LENGTH, CONCATENATE, KEYBD)
   Bit Strings (BITSET, BITCLR)
   System Data (TIME, DATE)

6. Input/Output (OPEN, CLOSE, READ, WRITE)

7. Initialize Digital Logic (ASSIGN LGC, CLEAR LGC)

8. Initiate Digital Burst (BURST)

9. Digital Testing (SET LGC, TEST LGC)

10. Flow Control, Burst:

    Branching (GOTO)
    Looping (LOOP, LOOPIF, EXITIF, OR)
    Subroutines (GOSUB, SUBROUTINE/RETURN/END SUBROUTINE)
    Exiting (BREAK)
    Enter or exit high-speed section (FAST, END FAST)
     (Systems with SpeedPlus option only)

11. Terminate Digital Burst (END BURST)

12. Terminate Program (END)

13. Subroutines, Non-burst (SUBROUTINE/RETURN/END SUBROUTINE)

The purpose of this section is to familiarize the reader with the available test language statements, and to provide a basic understanding of their function and how they may be used in a typical program. Detailed reference information on each statement can be found in Section 6.

3.2 INITIALIZING THE TEST PROGRAM

-----DECLARE-----

Before any other statements appear in the program, all variables and subroutines introduced by the programmer in the main body of the program must be identified by a series of DECLARE statements. These declarations tell the translator what user-defined entries to expect when compiling the source test program. If you forget to DECLARE a variable or a subroutine, you will get an error message when attempting to translate the program.

Each DECLARE statement can identify one variable, for example:

```
DECLARE RVAL;
DECLARE CSTRING DSTRG(20);
```

where RVAL is a floating point variable (default variable type) and DSTRG is a character string variable up to 20 characters long;

or one subroutine with its arguments, e.g.,

```
DECLARE POTADJ(CSTRING VAL(8), FLOAT RVAL);
```

where VAL and RVAL are arguments used by subroutine POTADJ.

or one externally installed and defined subroutine with its arguments, for example,

```
DECLARE EXTERNAL RSHIFT(BSTRING LATCHINPUT(16), CSTRING PLACE(1));
```

where LATCHINPUT is a 16-bit string and PLACE is a 1-character string, and RSHIFT is a program written, compiled, and linked on a system equipped to prepare programs written in languages other than 227X test language.

NOTE: In addition to user variables and subroutines, the system-defined fail vector described in Section 5 must also be DECLAREd as a bit-string variable:

```
DECLARE BSTRING FAIL(5);
```

As a convenience in digital testing, groups of digital pins may be DECLAREd and assigned a name by a DECLARE GROUP statement. The group name can then be used to specify the state of all pins in that group. An obvious application of this feature is in testing a functional module such as a register

or counter. For example, if the group of digital test pins 10-14 were defined by the variable name OUT, i.e.,

```
DECLARE GROUP OUT(10-14);
```

then the test statement

```
TEST LGC OH(OUT);
```

would test each of those output pins for a high state (as indicated by the Output High pin specification. Refer to the TST LGC description in Section 6.).

As another example,

```
DECLARE GROUP IN(1-5);
SET LGC IG(IN=B'01010');
```

would set pins 1-5 to the state specified by binary number 01010.

If the SpeedPlus option is used, circuit nodes may have special timing nails attached to them. These timing nails cannot be used for testing (driving and sensing) that node. A DECLARE NAILBIND statement can be used, however, to temporarily attach a separate test nail to the node for the duration of a test, then automatically disconnect that test nail after the test is complete.

3.3 ANALOG INSTRUMENTATION STATEMENTS

-----SET, TEST, MEAS, CLEAR device-----

The SET, TEST, MEAS, and CLEAR statements directed to the analog test devices are used to control the outputs and monitor the inputs of those devices and to connect them to the appropriate UUT pins. Refer to Sections 4 and 6 for a description of the test devices and the statements used to control them.

3.4 AUXILIARY TEST STATEMENTS

-----DELAY, PAUSE, NAILOFFSET, USE, INDICT-----

The statements in this category do not directly control the system test instruments but are closely associated with test execution.

The DELAY statement temporarily stops program execution for a specified time interval. For example, after turning off a UUT power supply, a DELAY statement is recommended to ensure that all capacitors on the UUT have time to discharge before the power supply is removed. The following statements would accomplish this.

```
CLEAR UPS(4);              /*Turn off power to the UUT*/
DELAY 20M;                 /*Wait 20 ms for capacitors to discharge*/
SET PIO(0) HRLY(OPEN 8);   /*Disconnect UPS(4) from the UUT*/
```

A PAUSE statement suspends program execution until the user presses the START/CONTINUE button on the keypad. The PAUSE statement would typically be used in any procedure requiring interaction with the user. For example, the statement sequence,

```
WRITE 'PRESS START/CONTINUE BUTTON WHEN READY';
PAUSE;
```

informs the user that no further action will take place until the START/CONTINUE button is pressed.

NAILOFFSET provides a simple way of relocating all subsequent nail specifications. It is especially useful in testing a pc board that contains several identical circuits. (FNAILOFFSET is used to relocated fast nails in the SpeedPlus option.)

For example, the UUT shown in Figure 3-2 has 6 identical circuits. The test fixture has been wired so that corresponding test pins for each circuit are eparated by multiples of 16 pins for a 2270 or 2271, or 128 pins for a 2272. In this example, pins 1, 17, 33, 49, 65 and 81 are connected to the same nodes in each of the circuits. Each time the test loop is executed, the specified pin list will be offset by the value given in the NAILOFFSET statement. When the test program has been repeated for all 6 sets of pins, testing is then terminated.

When a test statement fails, the user has numerous options available for responding to that failure (refer to Section 5). The USE statement is associated with those failure-reporting options. Specifically, the USE statement allows the programmer to define what default options will be taken whenever subsequent test statements fail. The following simple example,

```
USE FSUB = FAILSUB;
```

will cause a test statement to call subroutine FAILSUB automatically whenever it fails (unless specifically indicated otherwise in the test statement).

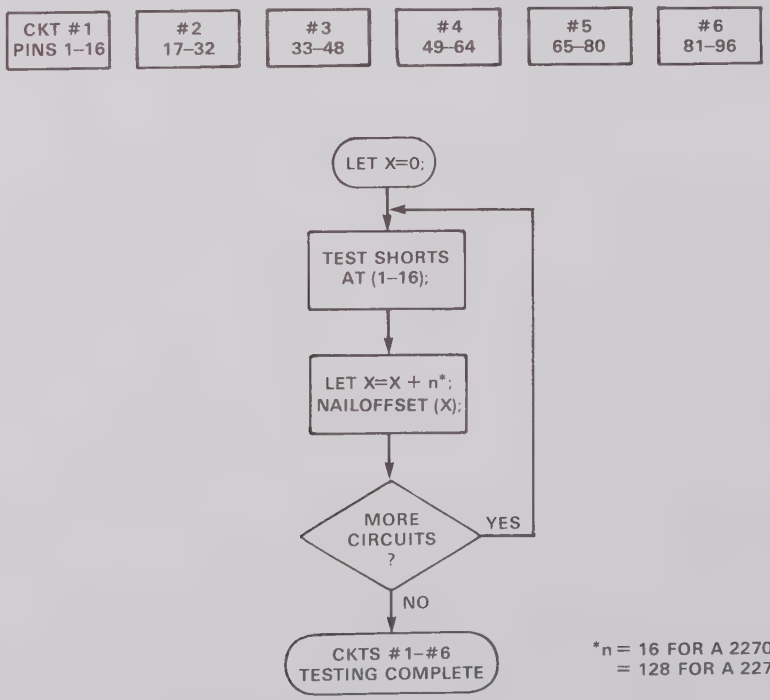


Figure 3-2. Example of NAILOFFSET.

To help the user customize diagnostic routines, an INDICT statement makes it possible to add digital ICs to the system probe list. The system directs the user to probe those specified ICs after the test program is complete.

3.5 FLOW CONTROL STATEMENTS (NON-BURST)

-----BRANCH, IF-THEN-ELSE, WHILE-DO, CALL, EXIT-----

NOTE

The 227X test language has 2 sets of flow control statements: those used inside a digital burst and those used outside a burst. These 2 sets are not interchangeable.

Flow control statements are used to alter the normal sequential flow of the test program. Outside a burst, the following flow control statements can be used:

Unconditional branching (BRANCH)
Conditional branching (IF-THEN-ELSE)
Looping (WHILE-DO)
Calling subroutines (CALL)
Terminating the program prematurely (EXIT)

The following simple examples illustrate their use.

3.5.1 Branching

The BRANCH statement causes a program jump to the specified location, regardless of previous conditions. For example,

```
        BRANCH TEST1;
        ...;
        ...;
TEST1:  ...
```

will cause the program to jump to the statement at location TEST1 immediately upon executing the BRANCH statement.

To make the transfer dependent on some previous event, the IF-THEN-ELSE statement can be used as shown in the following example:

```
IF V1<V2 THEN
[
WRITE .....;
...;
...;
];
SET MUX ...;
```

If the value of V1 is greater than V2, the program will jump to the SET MUX instruction. Otherwise, the program will execute the compound statement beginning with WRITE. Note that the THEN condition can be any simple or compound statement. Also note that the ELSE condition, not illustrated in this example, is optional and could be used to specify an alternate sequence of instructions if the specified condition (e.g., V1<V2) is not met.

3.5.2 Looping

Looping is handled by the WHILE-DO statement. For example,

```
WHILE N<100 DO
[
    '
statements
    '
LET N=N+1;
];
```

will cause the loop statements to be repeated as long as N is less than 100. The brackets identify the boundaries of the loop.

3.5.3 Subroutines

Procedures used frequently in a test program can be structured as subroutines. Non-digital subroutines and their arguments must be DECLAREd at the start of a program. Typical uses of subroutines are for failure reporting, interfacing with an operator, etc. They are placed after the main body of the test program (see Subroutines) and may be called from anywhere in the main body (except in a burst) by a CALL statement. (Refer to 3.11.4 for digital subroutines.)

The following example illustrates the subroutine structure.

```
DECLARE RSUB (CSTRING VAL(8), FLOAT RVAL);
    '
    '
CALL RSUB (VAL='R23', RVAL=10K);
    '
    '
END;

SUBROUTINE RSUB (CSTRING VAL(8), FLOAT RVAL);
...;
 ..;
...;
END SUBROUTINE RSUB;
```

Once the subroutine is DECLAREd, a CALL statement will temporarily transfer program control to that subroutine, execute the instructions, then return control to the normal program flow (i.e., to the statement immediately following the CALL statement).

Internal subroutines are the standard subroutines, written in the 227X test language, called during program execution, and stored in the test program file. External routines (described in Appendix C) are written in a high-level language other than the 227X test language and installed in the system as independent tasks. Load and unload subroutines are special user-defined internal routines that the system automatically calls during program load and program unload times.

Non-digital subroutines cannot be called from within a BURST.

3.5.4 External Subroutines

A facility for calling externally defined subroutines is provided in the 227X test language. External subroutines are not routines written in the 227X test language but, rather, independent programs written in MACRO-11 assembler or in a high-level language other than 227X test language. Since they are called in the same way as normal, 'internal' subroutines, they can conveniently extend the 227X test language by providing functions not otherwise available in test language (such as complex mathematical functions, bit string manipulations, or external interface drivers). An external subroutine, however, can only be prepared and installed at a facility having the DEC RSX-11M [1] Taskbuilder and appropriate tools for compiling and assembling the language in which the routine is to be written. The external routine must incorporate an interface protocol that mates with the 227X CALL protocol for external routines.

3.5.5 Forced Termination

An EXIT statement can be used to stop further UUT testing immediately, to reset the test hardware, to light the PASS or FAIL indicator based on test results and to exit from the test program.

3.6 DATA MANIPULATION STATEMENTS

---LET, SUBSTR, LENGTH, CONCATENATE, KEYBD, BITSET, BITCLR, TIME, DATE---

The statements, functions and operators in this category can be used to define program variables and to perform various operations on floating-point numbers, character strings and bit strings.

3.6.1 Variable Assignments

The LET statement is used to assign values to program variables. The assignment may be in terms of an actual value, an arithmetic expression or another variable. For instance,

```
LET S1='TRANSISTOR FAILURE';
```

assigns the character string 'transistor failure' to variable S1. The single quotation marks (') are used in the test language to delineate a character string. Note that S1 must have been previously DECLAREd as a character string variable at least 18 characters long.

Another example,

```
LET X=A*B/C+D;
```

will evaluate the arithmetic expression (A*B/C)+D and assign the results to the variable X.

[1] RSX-11M is a trademark of Digital Equipment Corporation, Maynard, MA 01754.

The arithmetic operators allowed in the 227X test language are

* * Multiply
* / Divide
* + Add
* - Subtract (or negate)

and are evaluated in the order shown.

3.6.2 Bit Strings

Individual bits in a bit string can be set or reset via a BITSET or BITCLR instruction. For example, if bit string PINLIST contained the following data,

```
1110111
```

a BITSET (PINLIST, 4) would set bit 4 to a one

```
1111111
   ^
   |
Set to 1----
```

and a BITCLR (PINLIST, 7) would reset bit 7 to a zero

```
1111110
      ^
      |
Reset to 0-----
```

The variable name PINLIST was used in this example to illustrate that bit strings can represent a scanner pin list in any statement used to control the scanner. A "1" closes the scanner relay represented by that bit position and connects the MUX output to that UUT test pin. A "0" opens the connection.

The fail vector described in Section 5 is also a bit string that can be manipulated via the BITSET and BITCLR statements.

3.6.3 Character Strings

The following examples (Figure 3-3, 3-4, and 3-5) illustrate how character strings can be manipulated by the test language. Assume that character string S1 has been defined as containing the 6 letters that spell G-E-N-R-A-D, i.e.,

```
LET S1='GENRAD';
```

To determine how many characters are in a particular character string, say, for example, in S1, you would use the LENGTH function. For example,

```
LET NUMB=LENGTH(S1);
```

would place the number 6 in NUMB, indicating that S1 presently contains 6 characters.

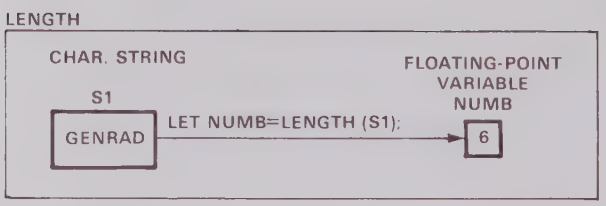


Figure 3-3. Length Function.

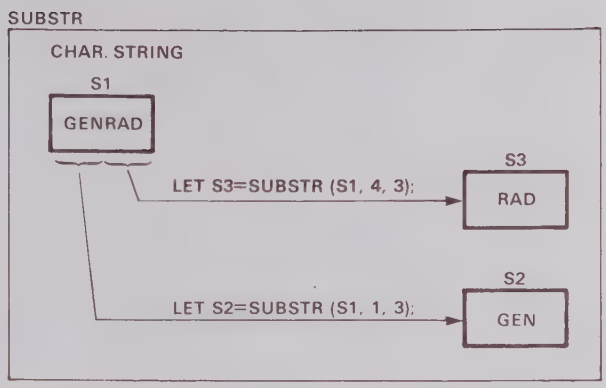


Figure 3-4. Substring Function.

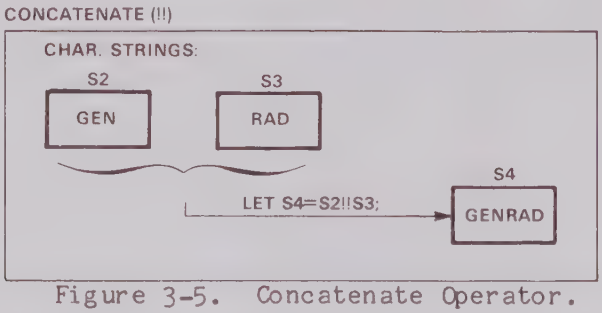


Figure 3-5. Concatenate Operator.

If you wanted to form substrings from portions of an existing character string, you would use the SUBSTR (substring) function. For example, executing the 2 statements shown in Figure 3-4 defines character strings S2 as 'GEN' and S3 as 'RAD.'

To combine substrings into a single character string (i.e., to append one string to another), you would use the CONCATENATE operator (!!), as shown in Figure 3-5.

A typical application of character string manipulation exists in the IEEE-488 Bus option. IEEE-488 device commands and messages are typically in the form of character strings that must be formed or analyzed by the test system.

The KEYBD( ) function can be used to sense whether or not a keyboard (or keypad) key has been pressed and to retrieve the character string resulting from pressing a key. A typical application would be to use the keyboard keys as user-operated sense switches, to control program flow. For example, in the following program sequence:

```
LOOPA:  ....;
        ....;
        ....;
        IF KEYBD( )='' THEN BRANCH LOOPA;
```

the instruction sequence starting at LOOPA would be repeated until any keyboard (or keypad) key was pressed. Then, program flow would continue in sequence.

3.6.4 System Data

The DATE( ) function is used to retrieve date and time information (in character string format) from the operating system. This information can then be written as part of user-generated messages.

The TIME( ) function is used to retrieve a floating-point number from the operating system, which specifies the number of seconds since midnight. This function is useful for calculating execution time for various program segments.

3.7 INPUT/OUTPUT STATEMENTS

-----OPEN, CLOSE, READ, WRITE-----

Input/Output statements are used to transfer data between the program and the various system devices such as the display/keyboard (TI:), the printer (LP0: or LP1:), or a disk file (DKn:, LBn:, ALn: SCn: or SY:). Typical data specified in data transfers include error messages, keyboard inputs, pin list files, etc.

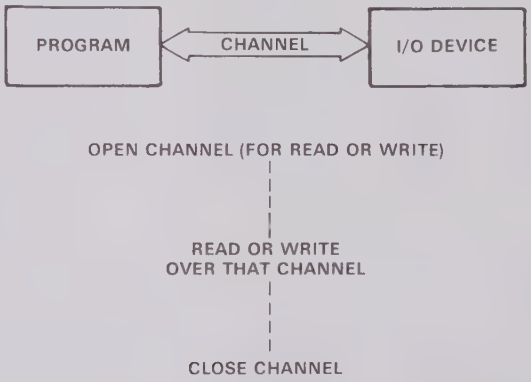


Figure 3-6.
Input/Output Flow.

All data transfers take place over I/O channels established by the user prior to issuing a READ or WRITE command (Figure 3-6). An OPEN statement activates the channel; a CLOSE statement deactivates it.

The keyboard and display are exceptions, since the system automatically opens channels for these devices at the start of the program and automatically closes them at the end. Special files MESFILE (para 5.3.3) and LOGFILE (para 4.6 in the System Operation manual) are also automatically opened and closed by the system.

Once a channel is OPENed, a WRITE command will direct the specified data to the designated output device (e.g., the display, printer, or a disk file), and a READ command will accept data from the specified input device (i.e., the keyboard) and assign it to the designated variable(s).

The READ and WRITE statements can also be used to convert data from one format to another (e.g., from ASCII to floating point). Refer to Section 6 for details.

3.8 INITIALIZING THE LOGIC

-----ASSIGN LGC, CLEAR LGC-----

The ASSIGN LGC and CLEAR LGC statements are the only digital statements that can be used outside a burst. ASSIGN LGC defines up to 2 sets of logic voltage levels and specifies what test pins are assigned to each set. CLEAR LGC resets the test hardware (driver/sensor system) used for digital testing. Refer to Section 4 for details.

3.9 INITIATE A BURST

-----BURST-----

All digital testing is performed in a high-speed burst mode. The sequence of statements that make up a burst starts with a BURST statement at the beginning and ends with an END BURST statement.

In addition to marking the start of the burst, the BURST statement also defines the following burst parameters: (1) the timing associated with each digital test step (i.e., Inter-Test Time (ITT) and Inter-Strobe Time (IST)), (2) the maximum time allowed for that burst (MAXTIME), (3) the state in which to leave the test hardware upon completion of the burst (CLEAR/ACTIVE), and (4) other failure-reporting options normally associated with instrumentation statements.

Systems having the SpeedPlus option can execute special high-speed sections within a burst. These sections are marked by the keyword FAST and end with END FAST. Instead of ITT and IST parameters, they use a larger number of clock and triggering parameters to specify timing and test synchronization (e.g., SET CLOCK, SIGNAL, CSYNC).

3.10 DIGITAL TESTING

-----SET LGC, TEST LGC-----

The actual setting-up and testing of the digital ICs is accomplished by the SET LGC and TEST LGC statements. These statements control the outputs of the digital drivers (SET LGC and TEST LGC) and monitor the inputs to the sensors (TEST LGC only). Refer to Section 4 for details.

3.11 FLOW CONTROL IN A BURST

GOTO, LOOP, LOOPIF, EXITIF, CR, BREAK
---GOSUB, SUBROUTINE, RETURN, END SUBROUTINE---

As previously mentioned, the 227X test language has separate sets of flow-control statements for use inside and outside a burst. The flow-control statements that are only allowed inside a burst are as follows:

Branching (GOTO)
Looping (LOOP, LOOPIF, EXITIF, OR)
Calling Subroutines (GOSUB)
Terminating the burst prematurely (BREAK)

3.11.1 Branching (Burst-only)

The GOTO statement is used for branching within a burst, either conditionally or unconditionally. For example,

```
        GOTO IC1;
        ...;
        ...;
        ...;
IC1:    ...;
```

causes an unconditional branch to location IC1.

However, the sequence,

```
        TEST LGC ...;
        GOTO IC1 FAILS;
        ...;
        ...;
        ...;
IC1:    ...;
```

will cause a transfer to IC1 only if the TEST LGC statement failed. The keywords PASSES and FAILS can be used to specify the condition for transfer.

3.11.2 Looping (Burst-only)

Looping within a burst may be useful for various phases of IC testing, for example, for initializing a counter, for sequencing through a memory, etc. Loop statements within a burst are bracketed by the keyword LOOP in the first test statement of the loop and the keyword END LOOP in the last test statement of the loop. In a FAST burst section (described in para 3.11.3), the keywords FLOOP and END FLOOP must be used.) A maximum number of iterations may be specified in the LOOP statement and additional loop/exit conditions may be designated by the LOOPIF, EXITIF, and OR modifiers. For example,

```
BURST ....;
...;
...;
...;
LOOP TEST LGC ...;
    '
    '
(loop statements)       With optional modifiers
    '                   LOOPIF, EXITIF, OR
    '
TEST LGC ... END LOOP;
...;
...;
...;
END BURST;
```

The loop modifiers can specify loop or exit conditions in terms of particular pin states. For example, in the following loop:

```
LOOP=100 SET LGC IH(1);
SET LGC IL(1);
EXITIF OL(2,3,4) END LOOP;
```

pin 1 will be pulsed high, then low, then high again until either output pins 2,3,4 are all low or until input pin 1 has been pulsed 100 times, whichever comes first. A typical application of this loop might be to reset a counter to zero and to quit if the counter hasn't reached zero after 100 counts.

3.11.3 Fast Section (with SpeedPlus option only)

Systems having the optional SpeedPlus subsystem recognize a special set of keywords that are used to initialize the subsystem's timing and synchronization parameters and to mark blocks within a burst for execution at high speed (up to 5 MHz) on the subsystem. Test statements within FAST sections are written with the usual TEST LGC and SET LGC commands (e.g., IH(), OI(), etc.). Subroutines (GOSUB) and branches (GOTO) are not permitted in FAST sections. Branching into, out of, between FAST sections, or over FAST sections is also not allowed. Other statements must precede the burst to define the timing parameters that will be used when FAST sections are executed.

FAST sections can only perform tests via driver/sensor nails that are part of the SpeedPlus subsystem (refer to para 6.4.4 for numbering of the SpeedPlus test nails). Other sections of a burst can use both the standard and the SpeedPlus driver/sensor nails.

In addition to driver/sensor nails, the SpeedPlus subsystem furnishes special clock and trigger nails that are used for test timing. These timing nails are used to aid and control the high-speed digital test but do not themselves execute burst statements. Consequently, TEST and SET LGC statements cannot reference these nails, either in a FAST section or elsewhere in a burst, as they have no standard testing capabilities. Analog and digital tests can only be performed at SpeedPlus timing nails by associating the timing nails with test nails via the DECLARE NAILBIND statement (described in Section 6.2).

At any point within a burst, a test can continue in the FAST mode; it reverts to normal testing after an END FAST statement. Before the burst begins, a CLOCK statement must define the timing and synchronization parameters that will be used in all of the FAST sections in the burst. The parameters include the basic test rate, the timing of drive and sense strobes (corresponding to ITT and IST in standard testing), trigger sampling times, and clock waveforms to be supplied to the UUT. Most of these parameters default to typical values, so that a SET CLOCK statement can be relatively simple, such as

```
SET CLOCK AT (CDA=3) F=1.23MEG;
```

This instructs the SpeedPlus subsystem to make available a 1.23 MHz clock signal at clock nail 3 whenever a burst is executed. It also coordinates (via defaults) driving, sensing, and triggering with this clock waveform. A burst using this clock might have the following form. In the first step the burst drives and senses some logic on the board at normal test speed. Then it enters a FAST section, where it supplies a clock signal and drives and senses some other logic at a high test speed. The 1.23 MHz clock is turned on during all bursts that follow the SET CLOCK statement. The clock nails, however, are disconnected by an END BURST and reconnected by the next BURST statement.

```
SET CLOCK AT (CDA=3) F=1.23MEG;

BURST
    IC (32, 33, 34)  OS (22, 23)   /* check some */
    IH (32, 33, 34)  OH (22, 23);  /* peripheral logic */

    FAST; /* start clock and test at high speed */
      IC (4016, 4116)  OS (4216)
      IH (4016) OH (4216)
        *  *  *
      IL (4016) OL (4216);
    END FAST;

    OI (23) IH (32, 33);           /* check the peripheral */
    IL (32, 33) OL (22, 23);       /* logic again */
END BURST;
```

Because the SpeedPlus subsystem allows the user to define test steps in much greater detail that does the standard test hardware, the CLOCK and FAST sections can be fairly complex. Refer to para 4.4 for general information about how the SpeedPlus system operates, and to para 6.4.5 for use of the SpeedPlus keywords listed in Table 3-1.

Table 3-1
KEYWORDS USED BY SPEEDPLUS FUNCTIONS

CDA	FAST	NOSYNC	TH
CDB	FLAG	OFF	TL
CDC	FLAGFAIL	ON	USING
CDD	FLOOP	PERIOD	VCDH
CLKGND	FNAILOFFSET	PHASELOCK	VCDL
CLOCK	HISLEW	S	VCST
CSYNC	LOSLEW	SIGNAL	VTT
DST	NINC	SYNC	
ED	NOPHASELOCK	T	

3.11.4 Digital Subroutine

Subroutines used in a burst must be defined within that burst and do not require an argument list. They are always local to this burst, and are unknown elsewhere in the test program. (In contrast, analog subroutines, described in para 3.5.3, are global to the whole test program.) The subroutine definition must begin with a SUBROUTINE statement and end with an END SUBROUTINE statement.

Subroutines are not permitted in FAST sections of bursts, that is, in sections meant to run on the SpeedPlus subsystem.

Burst subroutines are called by a GOSUB statement. The GOSUB, like the GOTO statement, may be either unconditional or conditional on the pass/fail status of the previous logic test.

```
BURST ....;
    '
    '
GOSUB SBR1          /* call the subroutine */
    '
    '
SUBROUTINE SBR1;    /* declare and define */
...;                /* a subroutine */
...;
...;
END SUBROUTINE SBR1;
    '
    '
END BURST;
```

3.11.5 Forced Termination (Burst-only)

A quick exit from a burst can be effected by a BREAK statement. The BREAK statement in a normal burst will immediately terminate the burst and either reset the test hardware or leave it unchanged, as specified within the BREAK statement itself. For example,

```
BREAK ACTIVE;
```

will leave all digital pins in their existing state. The BREAK statement in a FAST (SpeedPlus) section, transfers control to the end of the current FAST section.

3.12 TERMINATE A BURST

-----END BURST-----

The END BURST statement marks the end of the statement sequence that makes up a burst. The state of the test hardware following execution of a END BURST statement is determined by the CLEAR/ACTIVE specification in the associated BURST statement.

3.13 TERMINATE THE PROGRAM

-----END-----

The END statement must be used as the last statement in the main body of a test program (just before any subroutines). The function of the END statement is to inform the translator that it has reached the end of the test program.

3.14 SUBROUTINES (NON-BURST)

-----SUBROUTINE, END SUBROUTINE-----

All non-digital subroutines (used outside a burst) must be DECLAREd at the beginning of the test program and defined following the END statement of the test program. As shown in the following illustration, each subroutine begins with a SUBROUTINE statement listing the arguments to be referenced by that subroutine from the main program and ends with an END SUBROUTINE statement.

```
    '
    '
(main program)
    '
    '
END;
SUBROUTINE SUB1 (      );
....;
....;
END SUBROUTINE SUB1;

SUBROUTINE SUB2 (      );
....;
....;
END SUBROUTINE SUB2;
    '
    '
   etc.
```

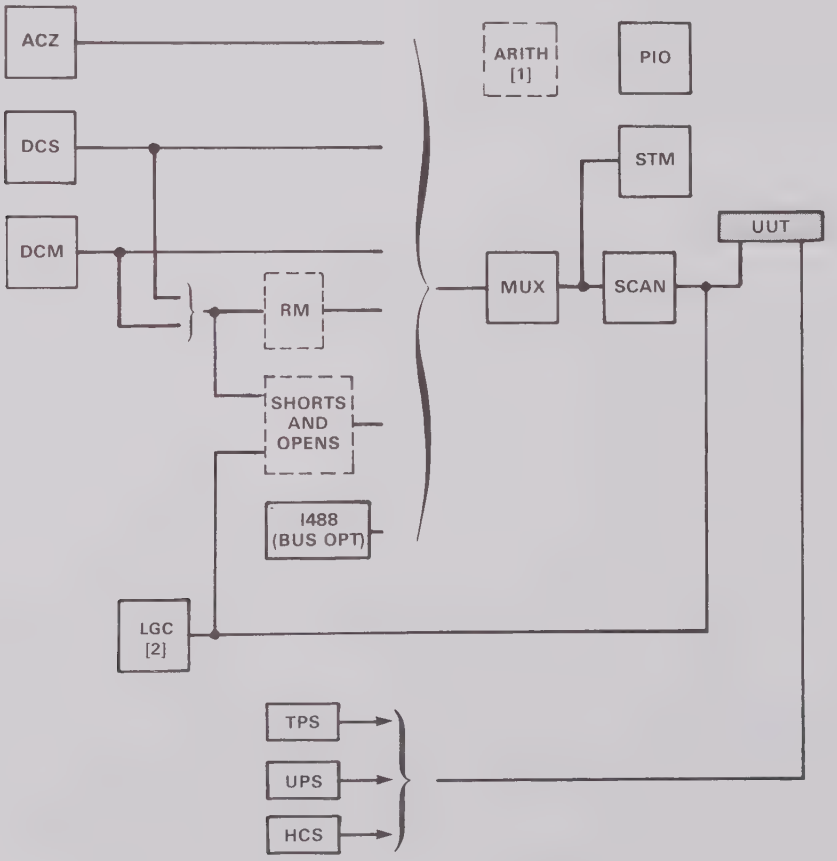
Testing the UUT–Section 4

4.1 HARDWARE AND SOFTWARE DEVICES

The Board Test Workstation provides the user with a number of hardware and software devices (Figure 4-1) designed for the efficient testing of a UUT.

Some of the devices are used directly to test the UUT; that is, they either produce the test inputs (stimuli) or measure the resulting outputs (responses) of the UUT. In this category are the dc source and measurement units (DCS and DCM), the ac impedance measurement unit (ACZ), the digital logic drivers and sensors (LGC), and any other similar user-supplied device that has access to the UUT via the IEEE-488 Bus option.

Other devices do not directly produce the test inputs or measure the outputs but contribute indirectly to testing the UUT. They include the



NOTE
[1]DASHED LINES INDICATE SOFTWARE MODULE.
[2]LGC MAY INCLUDE STANDARD DIGITAL TEST NAILS AND/OR SPEEDPLUS NAILS

Figure 4-1. System Test Devices.

instrument multiplexer (MUX) and pin scanner (SCAN) used to connect analog source and measure units to the UUT, the pin expander (PEX) used on the 2272 to select test nails for connection to the the scanner and driver/sensor outputs, the software drivers used for resistance measurements (RM) and continuity checks (SHORTS, OPENS), the UUT power supplies (TPS, UPS, HCS), a self-test module for verifying the operation of the system (STM), circuit boards with relay and TTL drivers and sensors for controlling special system and customer circuits (PIO), and a software module for testing arithmetic quantities (ARITH).

Detailed descriptions of each device can be found in Section 6 of this manual. The purpose of this section, however, is to introduce the reader to all the system instruments by presenting typical examples of how they are used to test the UUT and how they are controlled by the test language. At the end of this section is a summary of the general format for all 227X instrumentation statements and a brief description of the various parameters allowed in those statements.

To provide some order to this presentation, these system devices are grouped into the following categories:

1. Analog Test Devices
2. Digital Test Devices
3. SpeedPlus Test Devices
4. External Test Devices
5. UUT Power Supplies
6. Utility Devices

4.1.1 Analog Test Devices

AC Impedance Measurement Unit (ACZ)
DC Source (DCS)
DC Measurement Unit (DCM)
Instrument Multiplexer (MUX)
Analog Pin Scanner (SCAN)
Resistance Measurement Unit (RM)
Shorts Detector (SHORTS)
Opens Detector (OPENS)

All the devices in this group are standard 227X system devices used for testing analog (or hybrid) UUT pins. The ACZ, DCS and DCM are the source and measurement units used to force inputs and measure outputs. The MUX and SCAN are relay matrices used for connecting any source or measurement device to any analog (or hybrid) test pin. RM, SHORTS and OPENS are instruments simulated by software routines. They are programmed as if they were actual instruments, but in fact they use the DCS and DCM to perform their measurements. These simulated instruments simplify the programmer's job of setting up the source and measurement units, connecting them through the MUX and SCAN to the UUT, and then converting the results to resistance values or continuity indications. In addition, they perform the required calculations much faster than would the equivalent test language statements.

4.1.2 Digital Test Device

Digital Driver/Sensors (LGC)

Each digital (or hybrid) test pin is attached to a memory-controlled logic driver and sensor. The low-impedance drivers can force any pin to a logic high or low state, while the sensors can independently monitor the output of any pin to determine its state. A 1K memory (RAM) contains the test instructions, specified by the programmer and used by the driver/sensor controller for the high-speed burst testing of the IC chip.

4.1.3 SpeedPlus Test Devices

SpeedPlus Driver/Sensors (LGC)
Clock Drivers, Clock Sync (CLOCK)
Trigger Sensors (T)

The SpeedPlus option provides hardware and sofware facilities for carrying out in-circuit digital tests at up to 5 million test vectors per second. SpeedPlus bursts can contain up to 7K test vectors. The driver/sensor nails controlled by the SpeedPlus subsystem can also be used for standard digital testing. The subsystem can also automatically provide clock signals coordinated with the test steps and synchronized to a signal from the UUT. A triggering mechanism allows each test step to wait for a defined set of conditions before executing, so that test steps can be synchronized to UUT events such as bus cycles, read/write operations, or interrupts.

4.1.4 External Test Devices

IEEE-488 (or IEC) Bus Option (I488)

This option allows up to 9 external devices conforming to the IEEE Standard 488-1978[1] to be attached to the test system and to be operated remotely by the test language statements.

4.1.5 UUT Power Supplies

Triple Power Supply (TPS)
Universal Power Supply (UPS)
High-Current Supply (HCS)

In-circuit digital testing requires that power and ground be applied to the UUT. For this purpose, the system may have up to 3 types of program-controllable power supplies (TPS, UPS, HCS). The system also contains a special relay module or, in the 2272, special relays on each pin expander board. These relays switch a high quality system ground to the UUT under program control. The power-supply outputs can also be used to power-up any auxiliary test circuits that the user may have developed for a particular test.

---

[1]Copies of IEEE Standard 488 1978 Digital Interface for Programmable Instrumentation, Std. 488-1978, may be purchased from the Institute of Electrical and Electronic Engineers, Service Center, Department PB-8, 445 Hoes Lane, Piscataway, NJ 08854.

4.1.6 Utility Devices

Parallel I/O (PIO)
Self-Test Module (STM)
Arithmetic Test (ARITH)

The PIO and STM are both hardware modules. The PIO module consists of a number of program-controlled driver circuits for activating relays, TTL circuits, LED indicators, etc; and sensor circuits for reading the output state of the drivers and for monitoring TTL inputs. The STM module contains circuits and components of known values that can be used to check out and verify the operation of the various system test devices.

The ARITH unit is a software module that tests arithmetic quantities not directly measured by an instrument (e.g., the gain of a transistor). The arithmetic test is considered a system device because if it fails, the resulting action is the same as that of any other instrumentation statement.

4.2 ANALOG TEST DEVICES

4.2.1 General

A general introduction to the theory of analog testing on the 227X appears in the Test Set Preparation manual. The following paragraphs describe the analog test instruments (Figure 4-2), the Test Language commands associated with them, and how they are used in analog testing.

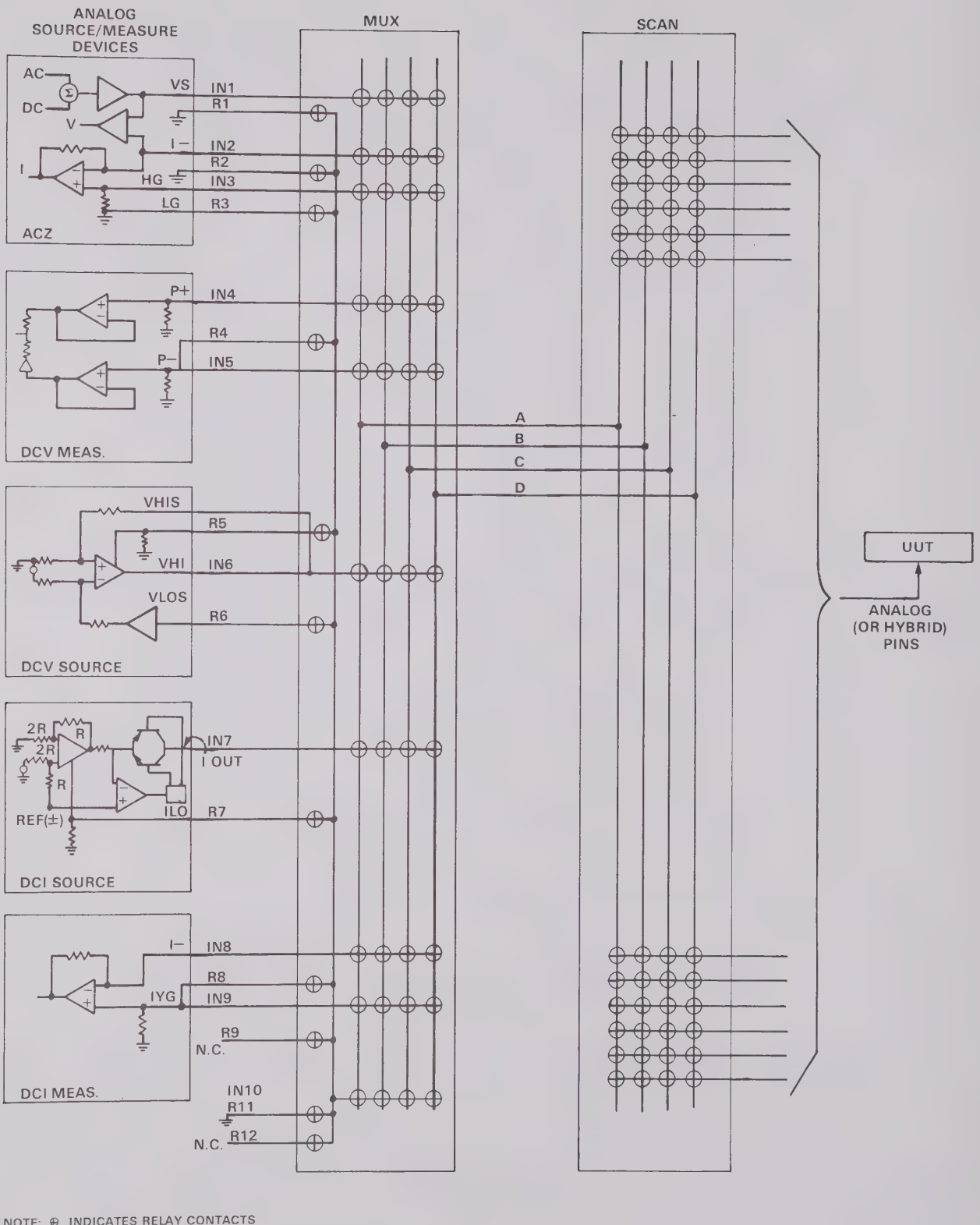
Typically the user of the 227X will not write test programs but will rather let the system's Automatic Test Generation (ATG) software generate them from a circuit description (.CKT) file. ATG derives its analog tests from an analog test library that provides a number of possible tests for each type of analog component. To measure resistors, capacitors, and inductors, ATG tries first to write a test using just 2 test nails. If it cannot get enough accuracy with such a test, it tries a test requiring 3 test nails. In unusual situations ATG may be unable to find a test that provides the accuracy required, or it may overlook circuit conditions that require special treatment. In these cases the user must devise an appropriate test strategy, instruct the system on how to connect the measurement instruments, and tell it how to measure the appropriate parameters.

4.2.2 Instrument Connections

The analog source and measurement devices (DCS, DCM, ACZ) can be connected to any UUT test pin via a program-controlled relay matrix, as shown in Figure 4-2. The relay matrix consists of an instrument multiplexer (MUX) that connects the selected instruments to any of 4 output channels (A, B, C, D) and a pin scanner (SCAN) that connects these MUX channels to the designated UUT pins.

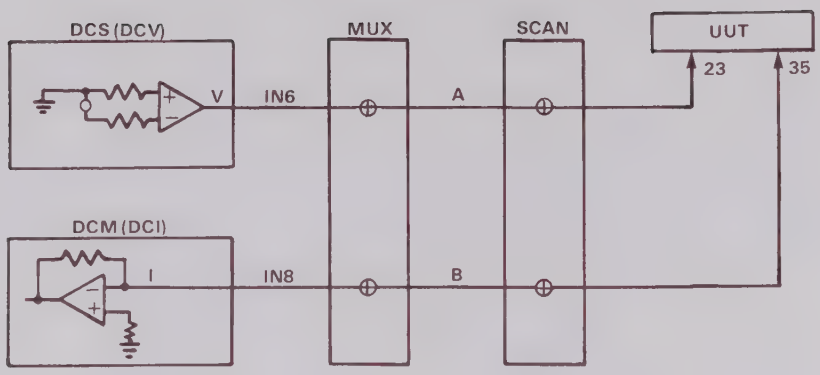
CAUTION

Reference lines on all instruments (except the DCM voltmeter configuration) must be kept within a few hundred millivolts of ground. For example, the current meter should not be connected in series with the high side of the DCV source. Failure to follow this guideline will cause errors.



NOTE: ⊕ INDICATES RELAY CONTACTS

Figure 4-2. Analog Test Devices.




```
SET MUX AT (CHA=6: CHB=8);
SET SCAN AT (CHA=23: CHB=35);
SET DCV DCS V=5.5;
MEAS DCI INTO IVAL;
```

NOTE: A diagram showing all test device connections to the standard MUX inputs is shown in Section 6, under the MUX description.

Figure 4-3. Typical DCV/DCM Connections.

In addition to the 4 output channels, the MUX also has a program-controlled reference bus that can be used to apply system ground to the reference (guard) input of any analog device or to any of the 4 output MUX channels. The MUX and SCANner are independently controlled by test language statements.

The simple example shown in Figure 4-3 illustrates how the test language is used to connect the dc source and measurement units to 2 separate UUT pins and how a voltage is forced at one pin and the resulting current measured at another pin.

The SET MUX statement connects MUX input 6 (which is the dc voltage source output) to channel A and MUX input 8 (which is the dc current meter input) to channel B.

The SET SCAN statement then connects MUX channel A to UUT pin 23 and channel B to UUT pin 35.

The SET DCV DCS statement causes the dc voltage source to develop an output voltage of 5.5 V, which is connected through the MUX and SCANner to test pin 23.

Finally, the MEAS DCI statement measures the current at test pin 35 and stores the value (in amps) in variable IVAL.

Note that these 4 statements have been kept simple to illustrate the basic principles. In practical applications, each statement might have several additional parameters specifying delays, current limit, failure reporting options, etc., each with its own default values. Refer to para. 6.3 - Analog Intrumentation Statements for details.

Note also that the circuit value measured in this test is a current expressed in amperes. To obtain the actual resistance value, the programmer would have to include an additional arithmetic statement (i.e., LET RVAL= VVAL/IVAL). Refer to para 6.3 for details.

4.2.3 Resistance Measurements

To simplify the programming of a resistance test or measurement, the test language contains a software-implemented resistance measurement driver. This driver still uses the DCS and DCM to force a voltage and measure a current, but it conveniently combines into a single statement, the DCS, DCM and SCANner pin specifications and the actual computing of the resistance value.

The TEST R (or MEAS R) statement enables the user to combine pin scanner connections and dc source parameters in the same statement and then automatically computes the resistance value from the source voltage and measured current. This computation is performed more efficiently by the TEST R and MEAS R statements than it is by equivalent arithmetic statements.

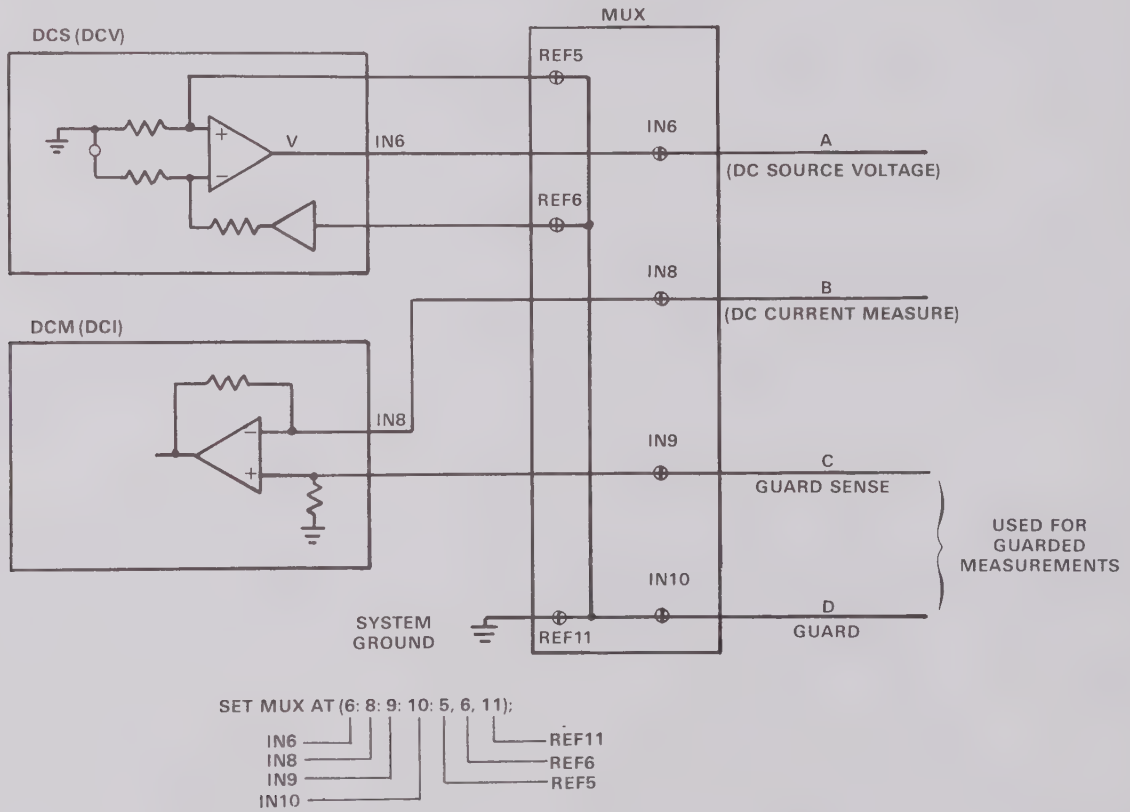


Figure 4-4. MUX Connections for Resistance Measurement.

Prior to using a TEST R or MEAS R statement, the DCS and DCM should be connected to MUX channels A, B, C, D, as shown in Figure 4-4.

The SET MUX statement implements the following MUX connections:

DCS voltage source to channel A
DCM current measure to channel B
DCM guard sense to channel C
Guard to channel D

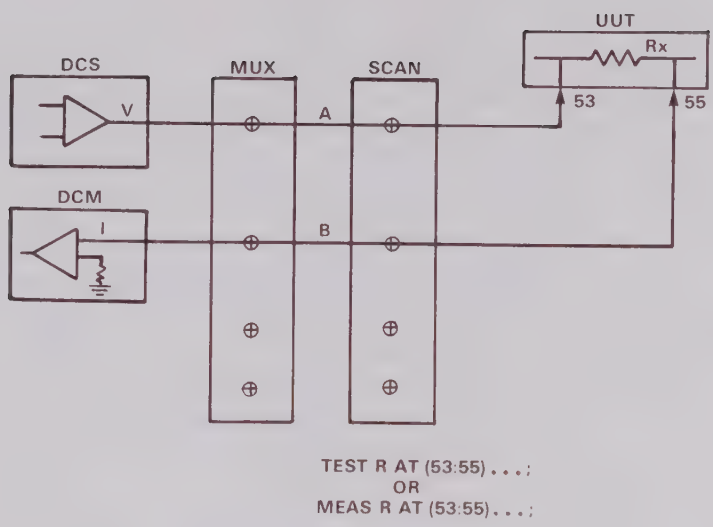


Figure 4-5. Typical 2-Terminal Resistance Measurement.

Once these connections are made, any number of TEST R or MEAS R statements can follow. A pin list in the TEST R or MEAS R statement then connects any of these MUX channels to any of the UUT pins. In the example shown in Figure 4-5, the statement:

TEST (or MEAS) R AT(53:55) . . . . ;

connects channel A (the voltage source) to test pin 53 and channel B (the current measure) to test pin 55, and leaves channels C and D unconnected. This is a simple 2-terminal, unguarded resistance measurement, with a voltage applied to one node and resulting current measured at the other node.

If you wish simply to check a resistance against known limits, a TEST R statement can be used as follows:

TEST R AT (53:55) LO=4.5K HI=5.5K [BRANCH RFAIL];

This statement will measure the resistance between test pins 53 and 55 and check that the measured value lies between 4.5K and 5.5K. If it does not, then a branch to label RFAIL is taken. The TEST statement does not make the measured resistance value available to the user (except as failure variable FMEAS described in para 5.3). In order to obtain this value, a MEAS R statement must be used. For example,

MEAS R INTO RVAL AT (53:55);

would store, in variable RVAL, the measured value of the resistance (in ohms) between pins 53 and 55.

Note that, if the AT( ) parameter is not included in a TEST R or MEAS R statement, the last SCANner connection specified will be used for the present resistance measurement.

4.2.4 Guarding

The 2-terminal resistance measurement just described ignores the effects of any other component on the pc board. In many instances, other components surrounding the resistance (or other circuit parameter) being measured provide shunt paths around the component under test which can significantly affect the desired measurement. In those cases, the effects of the unwanted shunt paths can be temporarily eliminated (or at least reduced) by use of an electrical isolation technique known as guarding. Guarding can be implemented using either 3 or 4 MUX channels. In the following presentation, the 3-terminal measurement is described first to illustrate the guarding operation, then the 4-terminal measurement is discussed to illustrate the advantage of a fourth channel.

Guarded Measurement (3-Terminal)

Figure 4-6 shows the system connections for a 3-terminal guarded measurement. First, all nodes electrically adjacent to the component under test are connected together and tied to ground (0 V), which is the guard voltage. Next, a voltage is applied to node A and the resulting current measured at node B.

Z1 and Z2 represent the equivalent impedances from node A to guard and node B to guard, respectively. Note that, because of the grounded '+' input to differential amplifier A2, the '-' input is constrained to 0 V (virtual ground). As a result, there is no potential difference across Z2, because one end of Z2 is at 0 V, due to the guard connection and the other end of Z2 is also at 0 V because of the virtual ground at the amplifier input. No current, therefore, flows through Z2, and all the current through Rx flows through node B and into the DCM unit. Due to the high input impedance of the DCM amplifier, this current also flows through the feedback resistor R(GAIN), developing a measurable output voltage (E3) in proportion to the current through Rx.

At node A, the DCS voltage source develops a known voltage E1. Because E2 is at ground potential, voltage E1 is applied directly across the unknown Rx. Voltage E1 is also applied across Z1, but the resulting current flows to ground because of the guard connection. Thus, the applied voltage E1 can be divided by the measured current Ix to obtain the value of the unknown resistance Rx.

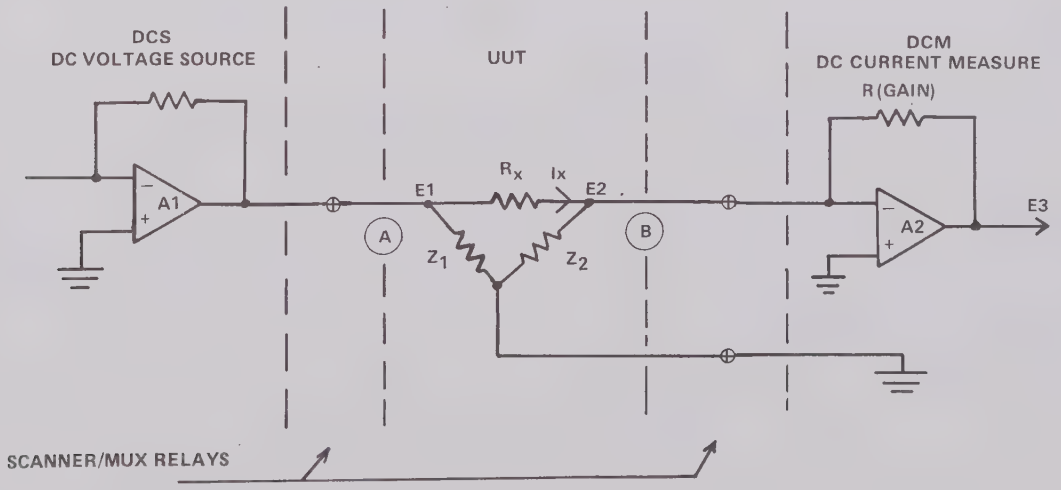


Figure 4-6. 3-Terminal Resistance Measurement.

Guarded Measurements (4-Terminal)

The source current shunted through Z1 during a measurement also flows through the SCANner and MUX relay contacts to system ground. If this current is significant, which it can be if several unwanted shunt components are tied together to a common guard, then a proportionately significant voltage drop can be developed across these contacts. As a result, a measurement error is introduced because the guard terminal, being above the ground reference, causes some current flow through Z2, thus affecting the current reading of the ammeter.

To counteract this source of error, tests written by the Automatic Test Generation software (ATG) use the 4-terminal measurement scheme shown in Figure 4-7. The reference input to the DCM amplifier is tied to the guard terminal at the SCANner rather than directly to system ground. This causes that guard input to the DCM to float above ground in proportion to the error voltage developed across the scanner relay contacts. This effectively reduces the error current into the DCM. Four-terminal impedance measurements using the ACZ are performed in this same manner.

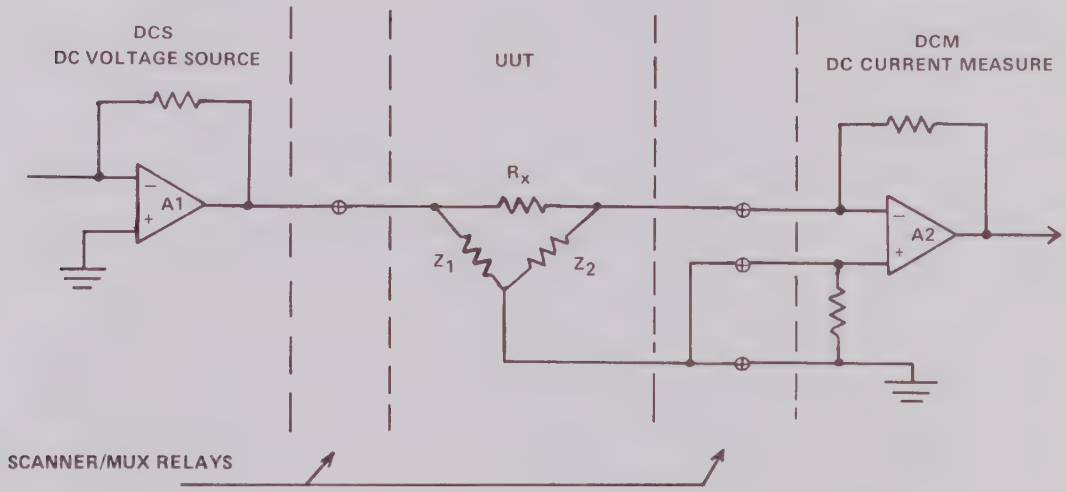


Figure 4-7. 4-Terminal Resistance Measurement.

An example of a 4-terminal guarded measurement, using the MUX connections previously shown in Figure 4-4, is shown in Figure 4-8.

The DCS is applied to one node of the component under test via MUX channel A, and the DCM is applied to the other node via channel B. Components adjacent to the component under test are tied together by the SCANner (test pins 18, 22, 37, 44) and connected to the DCM guard input via channel C and also to system ground via channel D.

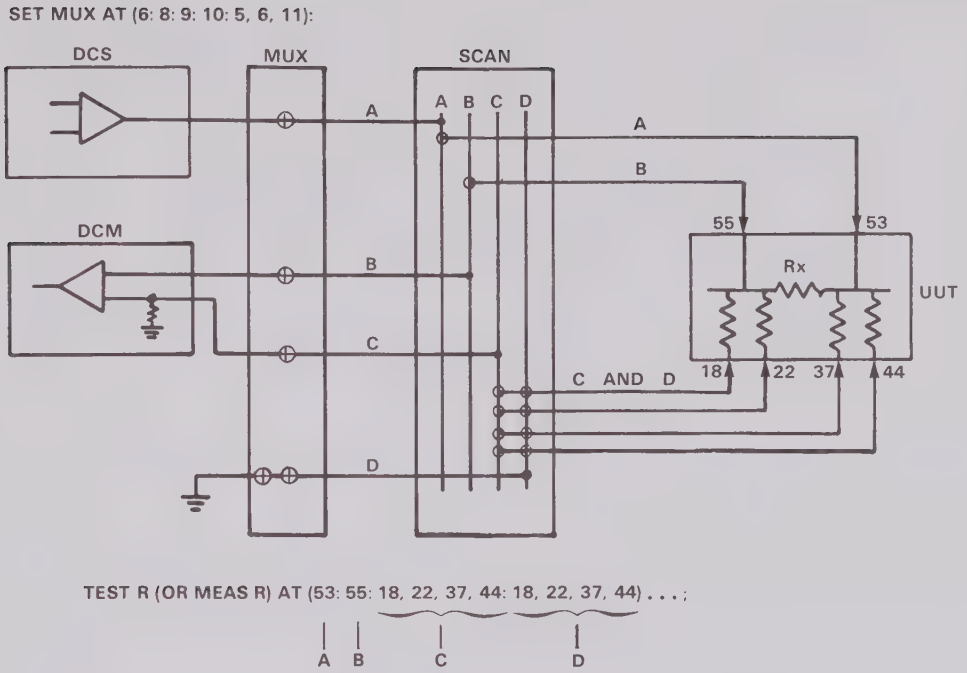


Figure 4-8. Typical 4-Terminal Resistance Measurement.

4.2.5 Impedance Measurements

The ac impedance measuring unit (ACZ) consists of an ac source and a phase-sensitive detector. Unlike its dc counterparts, the ACZ source and measurement units cannot be separately controlled. Impedance and admittance measurements are made by applying a known ac source voltage across the unknown impedance and measuring in-phase and quadrature (90 degrees out-of-phase) components of the resulting current and voltage.

The test language statements that control the ACZ are similar to those used to control the ohmmeter resistance measurement unit previously described. First, the ACZ must be connected to the MUX channels by the SET MUX statement

```
SET MUX AT (1:2:3:10:1,2,3,11);
```

As shown in Figure 4-9, this statement connects the ac voltage source to MUX channel A, the ac current measure to channel B, the guard terminal to channel C and analog ground to channel D.

A TEST or MEAS statement directed at the ACZ then connects the 4 channels to the appropriate UUT pins via the pin scanner and performs the indicated measurement. In the example shown in Figure 4-10, the statement:

```
MEAS CP INTO CVAL AT (CHA=33: CHB=35);
```

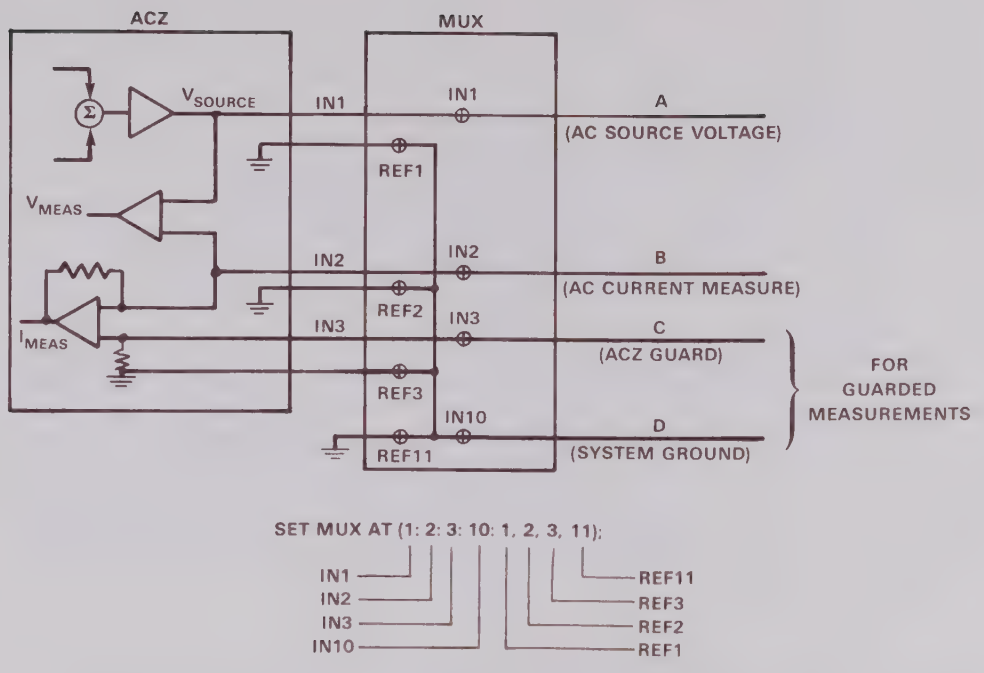


Figure 4-9. MUX Connections for ACZ Measurements.

will place the ac voltage source (channel A) on test pin 33, the ac current measure (channel B) on test pin 35 to measure the applied voltage and the resulting in-phase and quadrature current. The capacitance is then determined from these measured values, and the results stored in CVAL. Note that any of the ac functions allowed in an ACZ statement (i.e., RP, CP, LP, XP, Z, etc) can be determined from these measured voltages and currents.

Note that the keyword ACZ need not be included in a statement that measures an ac parameter (capacitance, in this case). As in the case of the R-measurement unit, MUX channels C and D are used only when making guarded measurements. Since the ACZ measures the voltage and both the in-phase and quadrature current, several circuit parameters can be MEASured or TESTed by the ACZ, including resistance (RS or RP), capacitance (CS or CP),inductance (LS or

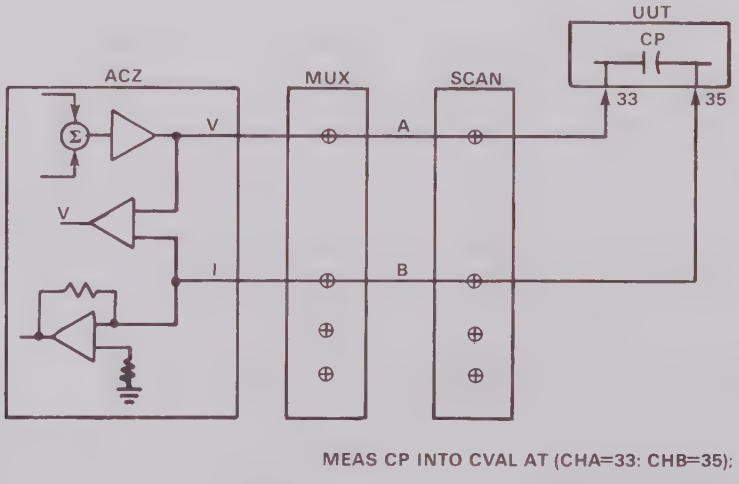


Figure 4-10. Typical 2-Terminal ACZ Measurement.

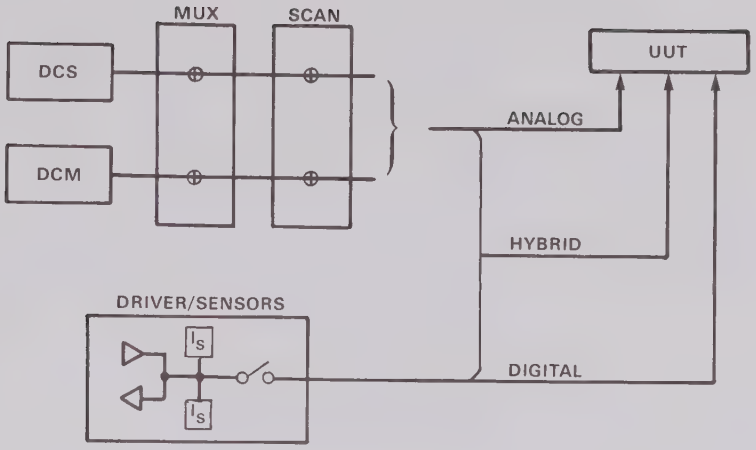


Figure 4-11. Continuity Testing of the UUT.

LP), reactance (XS or XP), impedance magnitude (Z), quality factor (Q) and dissipation factor (D).

4.2.6 Shorts and Opens Drivers

Both the SHORTS and OPENS drivers test for continuity between all nails in a specified nail list. The internal operation of both drivers is very similar. The major difference between the two is in what each of them considers a failure. To the SHORTS driver, activated by a TEST SHORTS statement, a short is considered a failure. To the OPENS driver, activated by a TEST OPENS statement, the lack of a short is considered a failure.

The actual continuity test itself is performed the same way by both drivers. As shown in Figure 4-11, hybrid (and 2270 analog) test nails are checked by the standard analog source and measurement units (DCS and DCM). Pure digital nails on the 2270, however, are checked by a current source/current sink attached to the output of each digital driver/sensor.

In the 2271 and 2272, all tested nails are hybrid and can be listed in the following format:

```
TEST SHORTS AT (tested-nails)
 (or OPENS)
```

On the 2270, a UUT may be connected to a group of analog (or hybrid) nails or to a group of pure digital nails. Thus, in a TEST SHORTS or TEST OPENS statement, the programmer must group the nails to be tested as follows:

```
TEST SHORTS AT (analog/hybrid: fast digital: slow digital)...
(or OPENS)          |            |              |
                    |            |              |
  Any nail connected ---         |              |
  to the scanner                 |              |
                                 |              |
    Any digital nail without a ------           |
    large reactive component                    |
    (e.g., capacitor) tied to it                |
                                                |
      Any digital nail with a large capacitor ------
      tied to it which would slow down the test
```

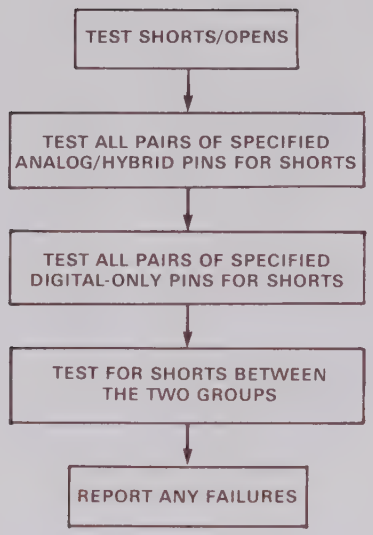

TEST SHORTS/OPENS
TEST ALL PAIRS OF SPECIFIED ANALOG/HYBRID PINS FOR SHORTS
TEST ALL PAIRS OF SPECIFIED DIGITAL-ONLY PINS FOR SHORTS
TEST FOR SHORTS BETWEEN THE TWO GROUPS
REPORT ANY FAILURES

Figure 4-12. Operation of SHORTS and OPENS Drivers.

If the test system is a 2271, 2272, or 2275 all listed nails will be considered hybrid, regardless of the nail grouping.

Briefly, continuity testing is performed as follows:

Using the source (DCS) and measurement (DCM) units, all pairs of hybrid (or analog) nails are tested for shorts. Default shorts threshold is 10 ohms, but other values can be specified in the statement.

On the 2270, all digital nails (i.e, those connected to driver/sensors only) are then tested for shorts. Since these nails are not connected to the MUX and SCANner, the DCS and DCM cannot be used. Instead, the driver portion of the driver/sensors is electronically disconnected and a current source/current sink arrangement at each driver output is used to test for continuity between nail pairs. The internal shorts threshold for digital nails is fixed at approximately 40±30 ohms. Digital nails designated as "slow" nails will cause the system to delay a little longer before deciding whether or not a short exists.

Next, on the 2270, after both the hybrid/analog and digital groups have been tested separately, the DCS/DCM in combination with the current source/sink on the digital nails are used to determine if any shorts exist between groups. The threshold for this test is the same as that for the digital, 40±30 ohms (approximately).

If any failures are detected (i.e., any unwanted shorts during a SHORTS test or unwanted opens during an OPENS test), a report of the failed nails is displayed for the user. The chart in Figure 4-12 summarizes the operation of the SHORTS and OPENS drivers.

4.2.7 More Difficult Measurements

All the measurement techniques described so far are available to ATG when it attempts to write an analog test for a component. However, in some circumstances ATG will not be able to produce a usable test. If, for example, it finds an analog component not described in the analog test library, it will

write no test. Because ATG is able to analyze only a limited number of circuit nodes, it may be unable to evaluate the relevant tests in the library adequately. In other cases, ATG will write a test into the test program that, to the user, appears feasible, but it will remove the test from the execution flow, rejecting it as too inaccurate. The user will then have to modify the test generated by ATG, or prepare a new test. Often a simple change will make the test usable. Knowing how to use the test instruments in difficult test situations, the user can sometimes devise measurements considerably more accurate than those produced by ATG.

Circuits with Large Guard Ratios

Beyond the inherent instrument inaccuracies, in-circuit analog measurements are generally limited by one other source of error: current in voltage sensing lines. In many measurement techniques a critical simplifying assumption is that a voltage meter or programmed source is able to sense the potential on a circuit node exactly. In fact, however, the wires, contacts, and relays that connect the instrument to the circuit node always have a small but finite resistance, and if current flows through this resistance, the voltage seen by the measurement instrument will be slightly different from that on the circuit node. In most cases this difference can be ignored; in some cases, however, it can seriously degrade what appears, at first glance, to be a good test.

This affects especially the guarded test ATG writes to isolate individual components on a circuit board. While guarding is meant to reduce measurement errors introduced by parallel impedances in the measurement circuit, in some circumstances it can actually introduce errors. These guarding errors can occasionally be larger than the errors guarding was meant to remove. To see why this should be so, we must examine more closely the 4-terminal guarded measurement circuit shown in Figure 4-8.

To perform a guarded resistance measurement, the dc voltage source DCS is programmed to source a specified voltage, usually 0.2 volt. The dc measurement unit, DCM, presents a virtual ground at its measurement input, B, so a current flows from MUX line A into test nail 53 and returns via nail 55, through line B, to the DCM. The guard currents from nails 18, 22, 37 and 44 return to system ground via line D. As explained in para 4.2.4, large currents can cause appreciable voltage drops across the scanner relays and mux lines. The guarded measurement can only be accurate if the voltage across every guarded impedance connected to the measure node is held exactly at zero. When ATG writes a guarded test, it eliminates a major source of potential error by connecting the current meter's ground reference to the guard nodes through a separate mux channel that carries no current.

A finer examination of the guarded measurement circuit will show, however, that there are times when this may not be sufficient. As shown in Figures 4-13 and 4-14, there are 2 other, small residual resistances that cannot always be ignored. Because these are in current-carrying portions of the circuit, the actual voltages on the current meter's inputs can be slightly different from the voltages on the connected nodes. It is the effects of these 2 residual resistances that are responsible for errors peculiar to guarded measurements.

The first of these effects results from the resistance represented as Rg1 in Figure 4-13, the contact resistance of the test nail and its connection to the scanner. This resistance can be as much as 0.1 ohm (or more, if the nail or circuit board is not clean); and it can begin to introduce significant

measurement errors in some situations. This occurs, in particular, when the ratio of the DUT to one or more of the guarded impedances is large. This ratio, called the guard ratio, is an important indicator of how successful a guarded measurement will be.

In Figure 4-13 the guard ratio is Rdut/Rmg (where Rmg represents the lumped impedance of all the guard nail connections in Figure 4-8). The value is 100/10 or 10. The guard ratio suggests that spurious voltages in the system, too small to affect the current in Rdut significantly, can cause currents 10 times as large in Rmg. In particular, the very small voltage that is developed across Rg1 will become significant. With the typical source voltage of 0.2 V a voltage of (0.2 V / 10 ohm) X 0.1 or 0.002 volts between the circuit node and the meter's ground reference. Since the current meter holds its current sink and ground reference terminals well within a millivolt of each other, node M can be regarded as connected to the ground reference line. Thus, some of the current from Rsg is diverted through Rmg into the current measure node, node M. At 0.002 volt, the magnitude of this current is 0.002 V/ 10 ohm or 0.2 milliampere. However, the current through Rdut is only 0.2 V / 100 ohm or 2 mA. The current into the measure node is the sum of both Imeas and Img, i.e., 2.2 mA. Because total measured current is greater than Idut, R appears smaller than it really is, 0.2 V / 0.0022 A or 90.91 ohms -- an error of nearly 10 percent.

A large value of Rdut/Rmg impairs measurement accuracy because it sends an additional, spurious current into the measure node from the guard node. Thus, the current meter measures more current than is actually flowing through the DUT, and the tester reports that the DUT's impedance is lower than it really is.

The ratio Rdut/Rsg can also be important. If Rsg is small compared to Rdut, the current flowing into the guard node is large relative to the current flowing into the measure node, and a relatively large voltage appears across Rg1. Even if Rdut/Rmg is small, enough current may flow into the measure node to make the measurement inaccurate. As a rule of thumb, values of Rdut/Rmg greater than 10 will impair accurate measurement. Large values of Rdut/Rsg will also affect accuracy, but typically less than the ratio of Rdut/Rmg. Poor (large) guard ratios result in measured impedances lower than the true values.

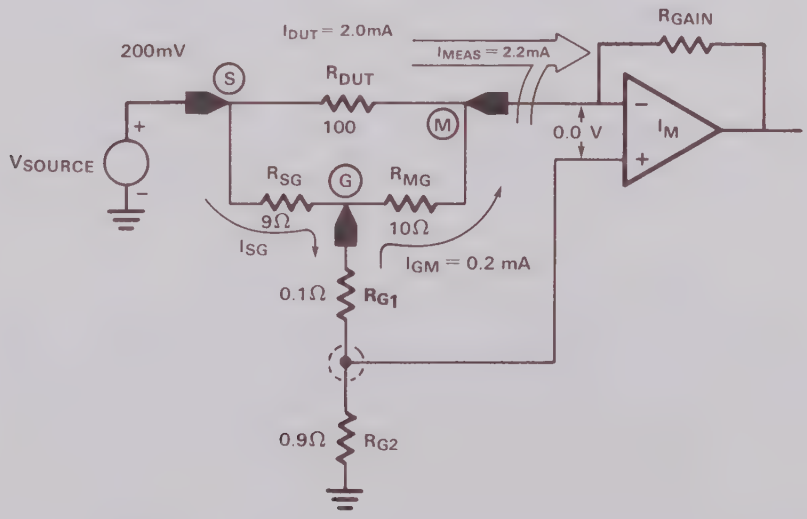


Figure 4-13. Causes of Error in Guarded Measurement.

Low-Value Guard Resistance

Another type of error can result from a low value of Rmg, independent of the guard ratio error. Let us assume that the value of Rsg in Figure 4-13 is large and that the current into the guard node is insignificant. Since there is little current into the guard nail, the voltage across Rg1 will be negligible. The current meter's ground reference will be very close to the voltage on node G and no error-inducing current should flow through Rmg. Upon closer consideration, however, we notice that the actual measurement circuit includes a resistance, Rm, between the measurement node and the current meter, as shown in Figure 4-14. (Rm includes both the relay-scanner resistance and the small intrinsic resistance of the current meter.) This means that the voltage on node M is slightly higher than at the current meter's input connection. The value of Rm is typically somewhere between 0.5 and 3 ohms, so if Rmg is small, the voltage at node M rises significantly above the current meter's ground reference, and a voltage appears across the guarded resistance, Rmg. Hence, some of the current through Rdut is diverted through Rmg into the guard node. Since the current meter registers less current than is actually flowing through Rdut, Rdut appears to be larger than it really is. A second rule of thumb for guarded measurements is that measurements of resistances lower than perhaps 50 ohms tend to be inaccurate, regardless of the guard ratios.

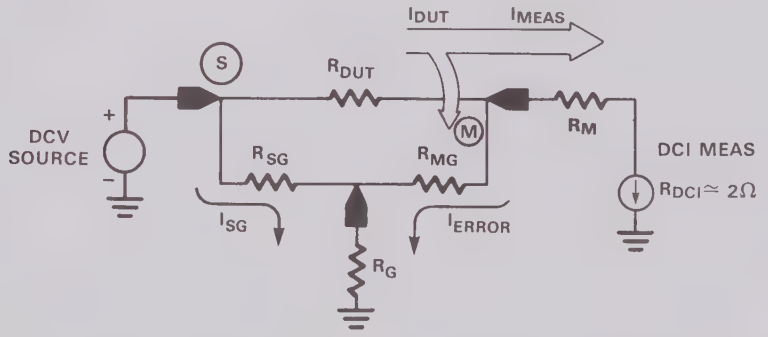


Figure 4-14. Errors in 4-Terminal Guarded Measurement.

It can, of course, happen that the error induced by Rmg -- which makes the DUT appear larger -- is exactly offset by the error from Rsg, which makes the DUT appear smaller. In this case, because ATG actually performs a detailed analysis of the circuit, it will correctly report that it has written a fairly accurate test, even though the circuit appears to have unfavorable guard ratios. ATG's analysis includes estimated worst-case values for Rm and Rg1. In addition, ATG usually surveys many or all of the various possible measurement configurations -- that is, the various ways the source, measure, and guard terminals can be connected. As it tries each measurement configuration, ATG calculates the effects of the system residuals and bases its estimated accuracy on these as well as on the accuracy of the measurement instruments. Thus, ATG is nearly always able to select the best test within its limited repertoire. (Refer to the Test Library Programming manual for information about how ATG selects and writes analog tests.)

When ATG is unable to find a test of the required accuracy, the test engineer must try to devise a better one. In cases such as the one shown in Figure 4-14 it is sometimes possible to obtain a more accurate measurement using a simple, 2-terminal configuration. Another useful approach is to short the DUT by closing scanner relays to a common multiplexer channel, measure the effective

residual resistance, and subtract this value from the guarded measurement. Since the short is not made on the circuit board but rather in the scanner, a small -- but usually insignificant -- portion of the residual resistance will not be subtracted (see Figure 4-15).

Low-Value Device-Under-Test

When the device-under-test, Rdut, is less than about 250 ohms, the system residual resistances can begin to cause errors. As shown in Figure 4-15, each connection from a measurement instrument to a DUT adds additional resistance, most of which is found in the multiplexer relays and connections. This additional series resistance is typically between 0.5 and 2 ohms, which can contribute a 1 percent error for a 200-ohm resistance, or 10 percent at 20 ohms. This error cannot be eliminated by guarding. ATG calculates its test limits by adding assumed minimum and maximum values of the residual series resistance (using the parameters RHIRESID and RLORESID, explained in Section 6 of the Test Set Preparation manual), so that resistors of correct nominal value pass the resistance test regardless of residual values. (Note, however, that a MEAS R INTO statement reports the actual measured value, without subtracting residuals.) Calculating test limits this way does not, however, allow ATG to produce a test more accurate than the uncertainty in the residuals.

The Kelvin measurement shown in Figure 4-16 is a commonly used solution to this problem when guarding is not required. It uses the programmable current source to force an accurately known current through the DUT. A high-impedance voltmeter connected directly to the DUT measures the voltage across it. Because virtually no current flows in the voltage sensing lines, the voltage seen by the meter is exactly the same as the voltage at the measurement nodes, and the accuracy of the test is limited only by the accuracy of the instruments. A good test can be performed with just 2 test nails by making the 4-terminal connection in the scanner; the small voltages developed across the test nails can usually be ignored. However, below about 25 ohms, separate test nails should be provided for current and voltage sense, so that the voltmeter reads directly from the DUT, as in Figure 4-16.

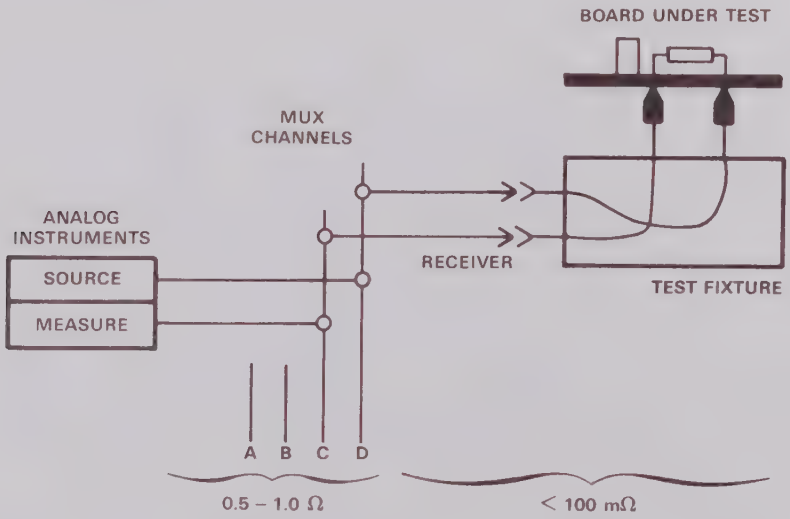


Figure 4-15. System Residual Resistances.

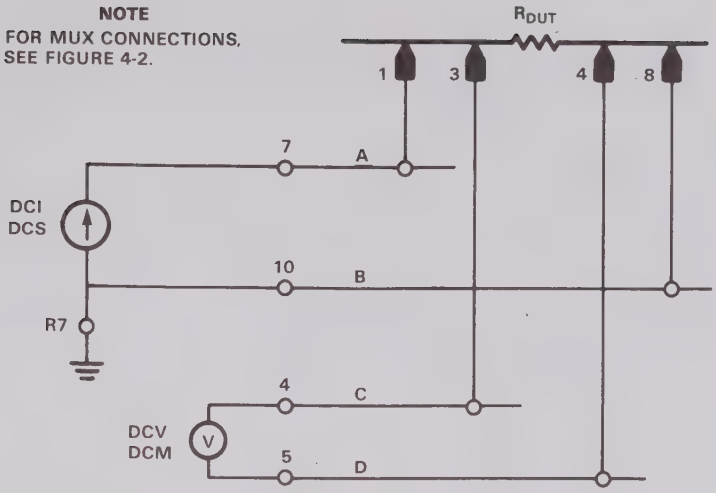


Figure 4-16. Unguarded Kelvin Measurement.

To prevent the current source from trying to force current to a defective or missing DUT, the first step in this measurement must be a check for a DUT of approximately correct value.

CAUTION

The DCS module in current-force mode can apply high voltages that can damage circuit board components. Always precede the statement SET DCS DCI with a test for approximately correct resistance value; execute the DCS statement only when the resistance is acceptably low. Use this test only for unguarded measurements.

Since only the resistance seen by the DCS needs to be checked, a 2-terminal resistance test is adequate.

The Kelvin connection can be provided by connecting the voltmeter to the nails through one pair of scanner relays, and the current source through another; for example, with the statements

```
SET MUX AT(CHA=7: CHB=10: CHC=4: CHD=5: REF=7,11);
SET SCAN AT(CHA=1: CHB=2: CHC=3: CHD=4);
```

These commands connect the dc current source (lines IN7 and R7 in Figure 4-2) to nails 1 and 2 through mux channels A and B, and they connect the dc voltage measurement unit (lines IN4 and IN5) to nails 3 and 4 through channels C and D. The measurement is thus performed by measuring the voltage that results when a pre-determined, known current flows through the DUT.

The test first checks that the resistance is in the expected range with a TEST R statement. A suitable upper limit is

```
(Rnominal + 2 ohms) * 2 * Rtol.
```

For a 9.76-ohm, 1-percent resistor, this rule gives an upper limit of 12 ohms. If the resistance is acceptably low, the DCS then applies sufficient voltage to force a current of 20 mA. The expected nominal voltage is

```
Vnom = 0.1952 = 0.02 * 9.76
```

In these voltage and current ranges an additional 0.5 percent instrument tolerance should be added, so the acceptable voltage range is 0.1952 ± 1.5 percent or

LO = 0.192 HI = 0.198

These calculations of nominal voltage and error limits could, of course, be written as part of the test program (with LET and TEST ARITH statements), and performed as part of the test at run time. In the interest of testing efficiency, however, a production test should avoid using the computer to perform calculations repeatedly that could be done beforehand. Floating-point arithmetic at run time requires a number of numerical conversions that can make a test slower.

```
/* FIRST CHECK FOR LOW ENOUGH RESISTANCE */

   SET MUX AT(CHA=6: CHB=8: REF=5,6,8,11);
   TEST R AT(CHA=1: CHB=2) LO=7.0  HI=12.0
          [WRITE 'RESISTANCE OUT OF RANGE';
           BRANCH OUTOFRANGE;];

/* R IS IN RANGE SO DO KELVIN TEST */

   SET MUX AT(CHA=7: CHB=10: CHC=4: CHD=5: REF=7,11);
   SET SCAN AT(CHA=1: CHB=2: CHC=3: CHD=4);
   SET DCI DCS I=0.02;
   TEST DCV LO=0.1913 HI=0.1991
          [WRITE 'RES IS BAD';];
OUTOFRANGE:
```

As with ATG's conventional measurement scheme (Figure 4-5), a guard node must be added to a Kelvin circuit when parallel impedances must be removed from the measurement. Because the unguarded Kelvin measurement already requires all 4 measurement lines, a line for the guard is substituted for one of the voltage measure lines. The voltage measurement must then be made in two steps, connecting a single voltage measurement line to the DUT's 2 pins in turn. In addition, the current meter must be used rather than the DCS to determine the DUT current, since not all of the supplied current flows through the DUT. Because the voltage rather than the current source mode is used, the 2-terminal check for in-range resistance is not necessary in this test. The guarded Kelvin measurement shown in Figure 4-17 could be programmed as follows:

```
/*SHUT DOWN DCS FOR RELAY SWITCHING */

   SET DCV DCS V=0.0;
   SET MUX AT(CHA=6: CHB=8: CHC=4: CHD=10: REF=4,5,6,8,11);
   SET SCAN AT(CHA=6: CHB=8: CHC=4: CHD=10);

/* MEASURE CURRENT AND V1 */

   SET DCV DCS V=0.2;
   TEST DCV DCS
          [WRITE 'DCS CURRENT LIMIT ERROR';
           BRANCH COMPLIERR;];
   MEAS DCV INTO VOLT1;
   MEAS DCI INTO IVAL;
```

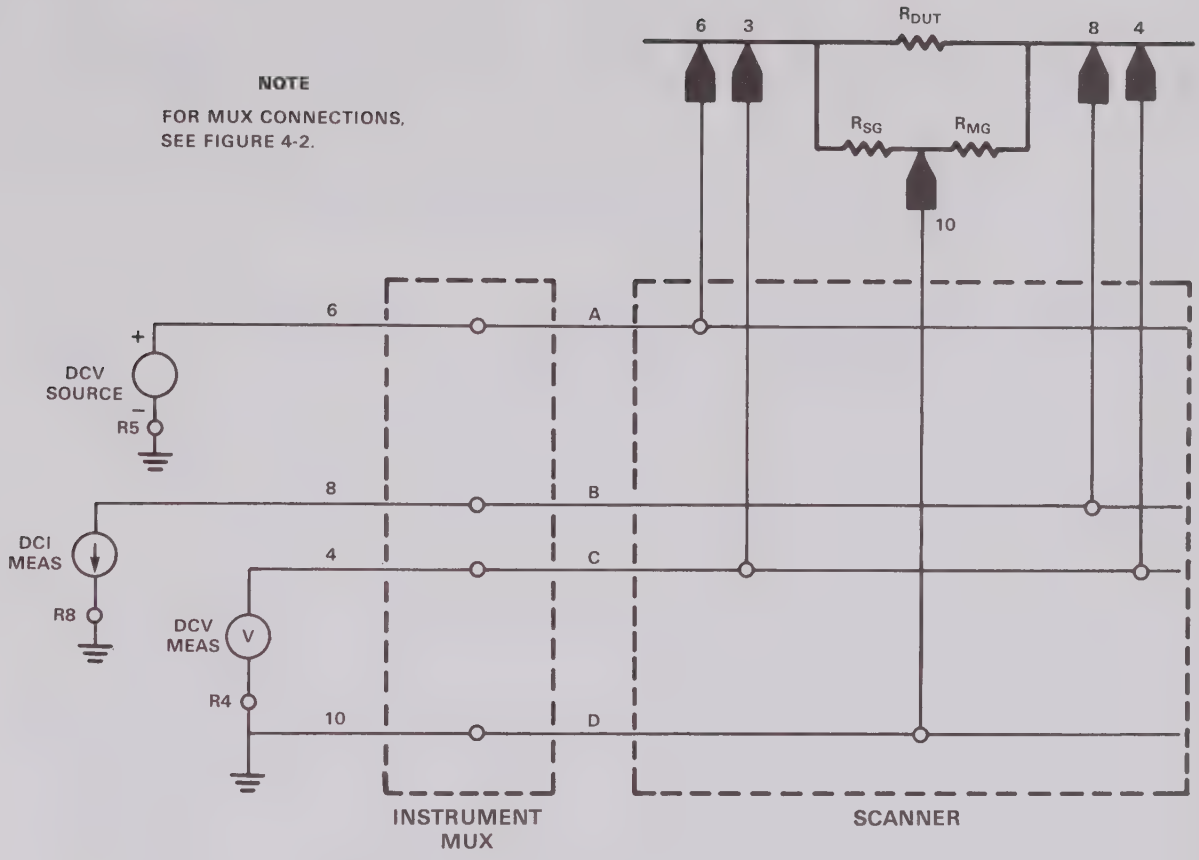


Figure 4-17. Guarded Kelvin Measurement.

```
/* SWITCH VOLTMETER TO OTHER DUT PIN; MEASURE V2 */

    SET SCAN AT(CHA=6: CHB=8: CHC=3: CHD=10);
    MEAS DCV INTO VOLT2;

/* COMPUTE R AND PERFORM ARITHMETIC TEST */

    LET VMEAS = VOLT2 - VOLT1;
    LET RMEAS = VMEAS/IVAL;
    TEST ARITH RMEAS LO=9.766 HI=9.947 [WRITE 'RES IS BAD';];

    COMPLIERR:
```

This program segment first turns off the DCS so that MUX and SCANner relays will not be switched while carrying current.

CAUTION

Do not execute SET MUX or SET SCAN statements when there is a possibility that current is flowing in a mux channel. Breaking a current may damage the relays.

It attaches the voltage source to one end of the DUT and the current meter to the other end. The voltmeter is referred to ground, so it measures the voltage relative to ground at nail 4. This value is stored in the variable VOLT1. Then the voltage source is turned off, the scanner disconnects the voltmeter from

nail 4 and connects it to nail 5, and the voltage (for the same current) is measured into VOLT2. The measured resistance is calculated as RMEAS = (VOLT2 - VOLT1) / IVAL.

Observe that this measurement, unlike the unguarded Kelvin measurement, cannot be performed without arithmetic calculations at run time.

Measurements of Large Resistances

Measurements of large resistances are usually limited in accuracy by the current meter's ability to measure the small currents that can be forced at low voltage. The resistance test, described in the Test Set Preparation manual, normally sets the voltage source to 0.2 volts to prevent diodes from conducting. At this voltage a 1-megohm resistor passes 0.2 uA, and the resulting current measurement contains a considerable amount of noise, which limits its accuracy to about 5 percent.

The measurement accuracy can be greatly improved by taking a series of 8 measurements and averaging. An averaged measurement, such as the one shown below, typically has an accuracy of 1 percent at 1 megohm and 0.2 V.

```
/* SET UP CONVENTIONAL RESISTANCE MEASUREMENT */

    SET MUX AT(CHA=6: CHB=8: CHC=9: CHD=10: REF=5,6,11);
    SET SCAN AT(CHA=1: CHB=2: CHC=3: CHD=3);

/* MAKE 8 RESISTANCE MEASUREMENTS */

    LET SUM = 0;
    LET LOOPVAR = 1;
    WHILE LOOPVAR <= 8. DO
      [
        MEAS R INTO SUMTERM MAX=1.2MEG;
        LET SUM = SUM + SUMTERM;
        LET LOOPVAR = LOOPVAR + 1;
      ];

/* COMPUTE THE AVERAGE VALUE */

    LET RMEAS = SUM / 8;
```

The noise can also be reduced by raising the source voltage. This is, of course, only practical in cases where current will not be diverted into adjacent semiconductors. In addition, the DCS current limit of 128 mA must not be exceeded.

Nodes with Connected Reactances

A valid dc measurement cannot be performed until the measurement circuit is in a static condition, so when reactive elements are connected to the DUT, ATG calculates an expected settling time and inserts appropriate DLY and RDLY parameters in the MEASure statements. These ensure that the measurement reaches the required accuracy.

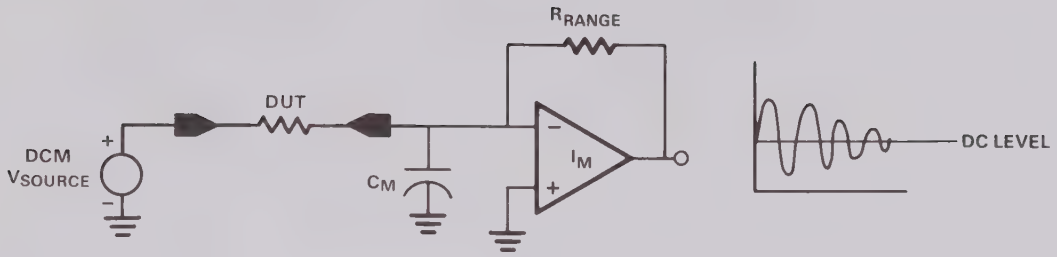


Figure 4-18. Capacitance on Measure Node.

A special kind of problem arises, however, with a capacitance on the measure, or current meter node, as shown in Figure 4-18. Thermal and rf-induced noise is present at all points in a circuit, including the capacitor Cm. Because a large capacitance can present a very low impedance at higher frequencies, the current meter responds as if large ac currents were being passed by the capacitor into the current measure node. (Actually, the compensation capacitor of the current meter amplifier and Cm form an active bandpass filter whose Q rises as Cm increases, so that the ac component confines itself to a narrow band of frequencies.) These ac currents add an error component to the dc measurement, and if ATG estimates the error will be too large, it rejects the test. ATG will generally try to avoid choosing a measure node connected to a capacitor, except where no other possibility exists. If ATG rejects such a test because of the capacitor-induced noise error, you can usually write a usable test by averaging a number of current measurements, using a loop similar to the one shown for measuring high resistances. The MEAS R statement cannot be used, however. Every time a resistance measurement is made, the voltage source turns on and off; this introduces pulse noise into the measurement circuit that makes the current measurement (under these circumstances) even more inaccurate. Instead, the voltage source should be left on, and a series of current measurements should be averaged, as shown below.

```
LET ISUM = 0;
LET LOOPVAR = 1;
WHILE LOOPVAR <= 8 DO
  [
    MEAS DCI INTO SUMTERM  FORCE MAX=0.15;
    LET ISUM = ISUM + SUMTERM;
    LET LOOPVAR = LOOPVAR + 1;
  ];
LET RMEAS = (0.2*8)/ISUM;
```

Impedance and Reactance Measurements

A single measurement unit (the ACZ) is used for measurements of inductance, capacitance, and impedance. Because only one instrument is involved, ATG is able to exploit the measurement capabilities more thoroughly than can be done for dc measurements, and there is generally less room for the test programmer to try other measurement techniques. In particular, Kelvin ac measurements are not available.

Occasionally a programmer may need to improve or modify an impedance measurement. Measurements of small capacitances and large inductances suffer from noise errors and can often be improved by the same sort of averaging technique shown above for large resistances. In some difficult measurement situations it will be found that the measured impedance consistently errs by a constant value; this can result from ranging or guarding errors. Small capacitors (less than 100 pF) may measure higher than expected because of parallel capacitances on the board (between foil tracks, component leads, etc). Inductors smaller than 50 uH or larger than 1000 H may require special tests to obtain high accuracy. With a good understanding of the measurement hardware it is possible to correct many of these errors, but generally the fastest method of correcting the test is to measure a component of known value in a known-good board, and use the measured value as the part's nominal value. Small bypass capacitors may be untestable when they are found in parallel with much larger supply capacitors that mask them. In such cases the bypass capacitors should be indicated with a NOTEST flagspec in the circuit description (.CKT) to shorten the ATG run.

4.2.8 Other Analog Tests

Tests for a number of analog devices must be written with special attention to their characteristics and circuit connections. A test written by ATG commonly contains default parameters that may not apply to the DUT. For these tests special parametric information must be supplied in the circuit description. Some devices, such as switches and relays, must be entered in the circuit description in a special way. Finally, there are inevitably devices for which no adequate test exists in the Analog Test Library -- for these devices the user must write a test and insert it in the test program. Once the test has been verified, it can be entered into the .ATO file so that it will be added each time a new test program is generated (refer to Test Set Preparation, Section 5). When tests already exist in the Analog Test Library for components of a module, a test for the whole module can be added to the Component Test Library (refer to Test Library Programing, Section 4).

Transistors

A library test is available for a single bipolar transistor. The default low limit for transistor gain is 5. A higher low limit can be specified in the Circuit Description (.CKT) file. In some circuits a missing transistor may produce a very high measured value of beta, which will allow the test to pass. To prevent this condition, an upper limit for beta should be specified. This value must be added as a high limit in the TEST ARITH statement of the test program file (.TPG) where the value of beta is checked, for example,

```
TEST ARITH CBETA LO=20 HI=1000 . . .
```

Darlington transistors should usually have the base-emitter flagspec specified as 2 diode drops; note, however, that some Darlingtons have internal resistors across the base-emitter junctions which may lower the effective forward voltage drop.

Diodes

The default forward voltage specified in the library test is 0.65 V at 20 mA, which is not appropriate for some diodes. ATG will write tests for Zener diode voltages up to 16 V; higher Zener voltages must be tested by using a power supply specially wired to the test node. A user power supply (UPS) of 20 V

maximum voltage or more must be available. Relays will be required in the test fixture to switch the supply voltage to the Zener test nails, since scanner switching is not available for power supplies.

An alternative approach is to check for forward conduction in the diode (which does not, of course, measure the Zener voltage). For default values and detailed descriptions of the ATG library tests, refer to the Test Library Programming manual, Section 8.

Transformers

Each winding should be described in the .CKT file as a separate inductor. The resistance of the winding should be determined empirically and entered as a DCR (dc resistance) flagspec. Components in series with the windings may turn out to be untestable. Examine components parallel to the windings; some may present guarding problems. Pay particular attention to damage that induced voltage from a step-up winding might cause when another winding is being tested.

Switches and Relays

A switch or normally open relay contact should be entered in the .CKT file as 2 EXTernals. A test for closure of the switch or relay contacts must be written manually and inserted in the .TPG file before the shorts test. For a switch, an instruction to the test operator should be included, followed by a PAUSE statement. The relay can be tested by activating the relay coil. Double and multi-pole switches should be described as a combination of a jumper and 2 or more externals.

4.2.9 Debugging Analog Tests

Sometimes it happens that ATG will write a test that fails during debugging (i.e., when the test is first run in the DEBUG submode of the DIAGnose mode, or monitor page). This is likely to happen when ATG is unable to predict correctly the behavior of semiconductors near the DUT, for example, when it has made an invalid assumption about a diode knee voltage or a transistor. The test engineer must then examine the circuit and the test to determine what changes ought to be made. Because changes to the test often require that test nails be moved or added, it is worthwhile first to look carefully for possible sources of error in the circuit description (.CKT) file and in ATG's circuit analysis.

ATG's 'intelligence' in writing a test derives from 2 sources. The first source is ATG's circuit analysis procedures, which furnish ATG with values for equivalent impedances between circuit nodes, taking into account various system residual impedances. The second source is the analog test library, which has lists of possible test instrument connections to the DUT, e.g., with or without guard connections or with the voltage source and current measure nodes reversed. (Refer to the Test Library Programming manual, Sections 2, 5 and 8, for more detailed descriptions.) Using equivalent impedances calculated by the circuit analyzer, the library procedures compute the accuracy of the test instruments under various conditions as well as the measurement errors introduced by the circuit configuration, extending to the nodes surrounding the DUT. ATG tries, in order, the test configurations it finds in the library until it finds one that provides a test with the required accuracy. (A more detailed description of the test development software appears in Section 7 of the Test Set Preparation manual.) If no test of sufficient accuracy is found, the last test analyzed is written into the test program, but a BRANCH to a label following it effectively

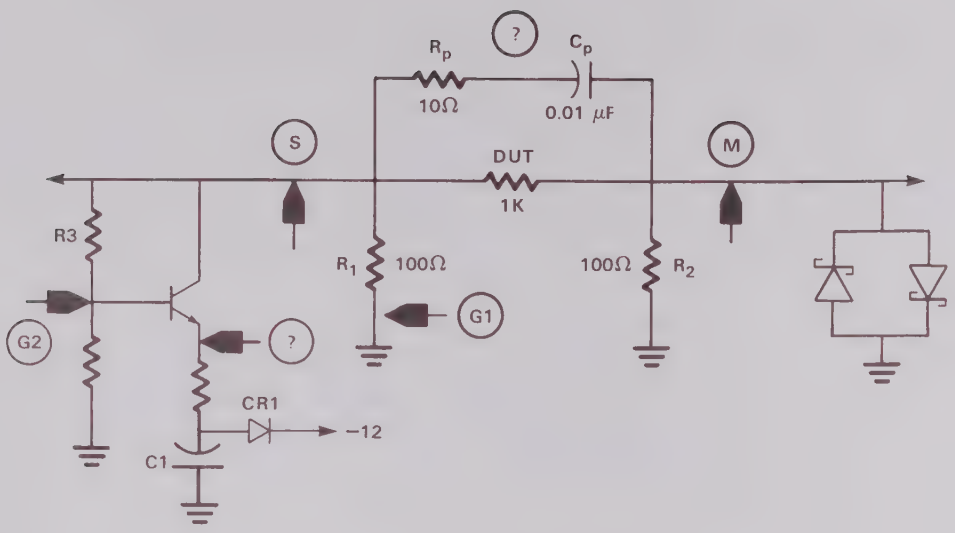


Figure 4-19. Guard Node Choices in an Analog Circuit.

eliminates the test program until it is corrected by the test engineer. (This kind of non-executing test is called a 'Form' test and can often be used, with a few modifications, as the basis for a usable test. In Release 6, form tests were written as program comments.)

Figure 4-19 shows a portion of an analog circuit with the nodes ATG has chosen for voltage source (S), current measure (M), and guard (G1, G2), in order to test the resistor DUT. We shall assume that the value of DUT is correct but that the test has failed during debugging; we shall then consider some of the conceivable causes of the failure, starting at DUT and proceeding to more remote nodes.

The first question to consider is whether ATG has made the best choice of measure and source nodes. We have seen that a capacitor like Cp at the measure node can impair measurement accuracy by introducing noise into the measure node. However, reversing the source and measure nodes will cause one of the Schottky diodes at the right to overload the voltage source. (Note, however, that ATG assumes a diode knee voltage of 0.65 volt unless told otherwise, and a missing knee voltage in the circuit description might have resulted in a faulty test with a source of 0.2 volt on the node adjacent to the diodes.)

Next we must examine ATG's choice of guard nodes. It has guarded R1 with nail G1, and R2 is thus also guarded by virtue of the common ground connection. Rp does not need to be guarded for a dc measurement because the capacitor eliminates it from measurement circuit. R3 has been guarded with nail G2, and it would appear that the transistor can be ignored, since the collector-base junction will be reverse-biased during the test. ATG, in fact, regards a transistor as 2 back- to-back diodes and thus assumes the transistor will not turn on during a test.

Because most analog testing is performed without power applied to the board, this is usually a safe assumption. The schematic suggests, however, that -12 V may have been applied at some point earlier in the test (to test an operational amplifier, for example). ATG assumes that a capacitor will discharge in the same time required to charge it, but it is evident that C1 might discharge much more slowly because it reverse biases CR1. If C1 was charged

earlier in the test program, then the transistor will conduct during the test of DUT. Because the voltage source is current limited, its voltage will fall below the programmed 0.2 volt, and the resistance test will fail. The error signaled by the voltage source's overload, or voltage compliance, condition cannot be identified unless the programmer explicitly adds a statement of the form

```
TEST DCV DCS [WRITE 'DCS CURRENT TOO HIGH';];
```

Alternatively, the error branch can be added to the SET DCS statement.

4.3 DIGITAL TEST DEVICES

Digital ICs are tested by driving the IC inputs with logic voltage pulses of sufficient drive capability to force those inputs to the desired state regardless of any interconnected signals. This "backdriving" technique is illustrated in Figure 4-20 by two interconnected type 7400 NAND gates.

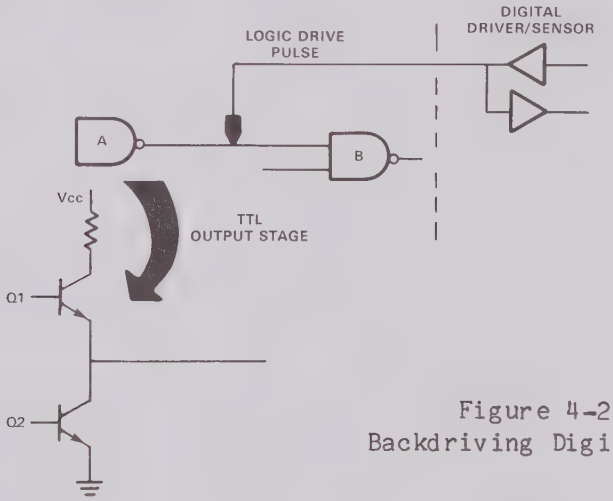


Figure 4-20.
Backdriving Digital ICs.

In Figure 4-20, a low (e.g., 0 V) output at gate A indicates that transistor Q1 in the output stage of that IC is turned on (i.e., in a low impedance state). To overcome this condition and force the input to gate B high (e.g., +4 V), a drive pulse is applied to the gate A output, with sufficient drive current to develop the required voltage across Q1. In typical applications, a current of 150 to 400 mA may be required to backdrive gate A and obtain a +4 V level at gate B. The drive pulse is kept short enough to prevent excessive power dissipation in the IC, but long enough to test the IC outputs for proper operation.

In addition to backdriving, most digital device tests contain an introductory section that disables other devices surrounding the component under test. This helps further to isolate it from its environment and prevents interference during the test. (Refer to Test Set Preparation, para 1.3.4, for a general discussion of in-circuit digital testing techniques.)

As shown in Figure 4-21, each digital (or hybrid) test nail in the system is connected to a memory-controlled driver/sensor circuit. The 1K RAM memory contains a user-defined (or ATG-defined) program that tests the IC in a high-speed burst mode. Prior to the burst, the system automatically loads the program segment defining the IC test into the driver/sensor memory. Then, on command from the driver/sensor controller, the stored test steps are executed at high-speed and resulting sensor data are stored 'on-the-fly' in memory, for further analysis after the burst is complete.

Hence, the first part of any digital test is an instruction to the test system to load the pattern of highs and lows that will be supplied to the UUT into the driver/sensor memory. Along with the statement, the test program usually supplies some parameters that apply to the burst as a whole. The segment begins with the statement BURST;, and it ends with END BURST;. The BURST statement often includes some information about test timing, as well; the term ITT (Inter-Test Time) defines the interval from one drive pulse to the next, and the term IST (Inter-Strobe Time) defines how long the tester should wait between driving and sensing. A typical burst has the form below; it may contain hundreds of digital test steps.

```
/* Define test step time of 6 microseconds */
/* Define drive-to-sense time of 2.5 microseconds */

BURST IST = 2.5 U ITT = 6 U;
    ¦
  <test steps>
    ¦
END BURST;
```

4.3.1 Driver/Sensor Programming

The logic drivers develop the logic inputs (stimuli) and the sensors monitor the logic outputs (responses). An internal protection circuit in each driver automatically reduces the drive current to tolerable levels if the driver remains on too long, thus protecting the IC from damage.

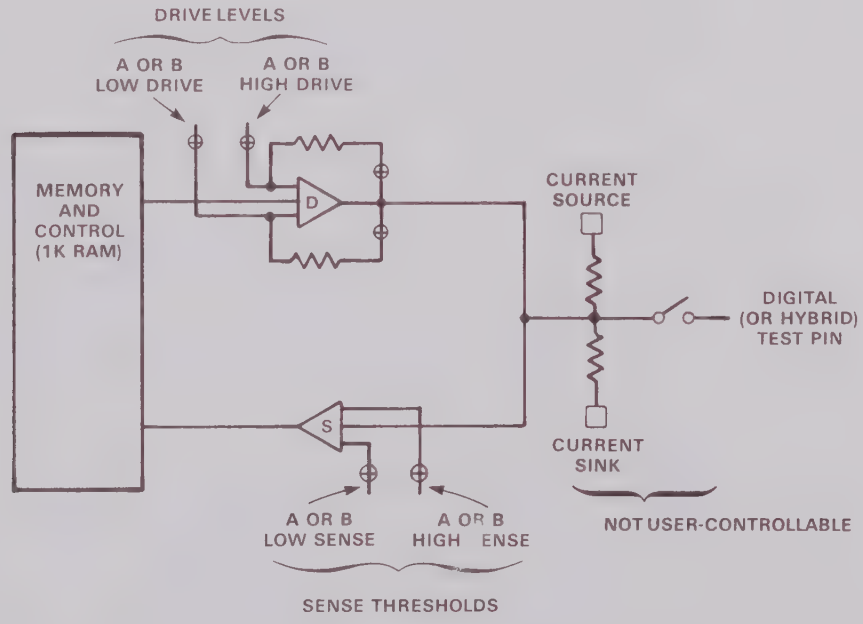


Figure 4-21. Driver/Sensor Circuit.

Under program control, each driver can be connected or disconnected (i.e., placed in a high impedance output state) from a test nail, and each sensor output can be enabled or ignored. Since the connect/disconnect and sense/ignore designations can be made independently of one another, each nail can drive a device's input pin, sense an output pin, or both drive and sense simultaneously (for self-test purposes).

In addition to connecting/disconnecting each driver from its test nail, test language statements can also define the logic levels for the test, force a connected driver output to a high or low state, attach a pull-up or pull-down resistor to each digital nail (for testing tri-state devices) or, by means of the shorts test described previously, use the current source/sink circuit connected to a driver output for testing continuity between digital nails. Similarly, the user can indicate the expected logic state of a signal into a sensor and enable or disable the output of that sensor, as desired.

The user has no direct control over the output (disconnect) relay that ties digital nails to analog nails in a hybrid configuration. The internal run-time system automatically closes this relay when a hybrid nail is being used for a digital test, and opens it when analog testing is taking place.

The logic (LGC) statements used to control the driver/sensors are described in detail in Section 6, but are also briefly presented here to demonstrate how the driver/sensors are used for digital testing.

The ASSIGN LGC statement can define 2 separate sets of logic voltages (set A and set B) and can assign either set to any digital nail. For example, if the UUT contained both TTL circuits (e.g., nominal LOW=0 V, HIGH=3 V) and RS-232 interface circuits (e.g., nominal LOW=-8 V, HIGH=+12 V), the following ASSIGN LGC statement might be used to set the voltage levels for the test and assign the appropriate set of voltages to each test nail.

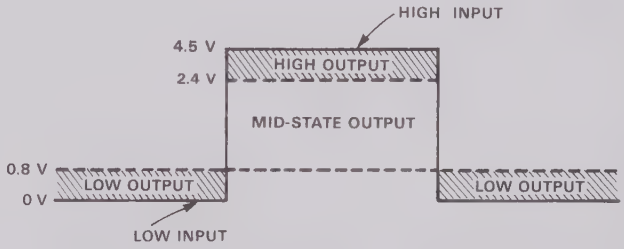
```
ASSIGN LGC  VIHA=4.5  VILA=0     VOHA=2.4  VOLA=.8   LVLA(1,3,7-9)
            VIHB=9.6  VILB=-5.5  VOHB=9.0  VOLB=-4.9 LVLB(2,4-6);
```

This statement defines 2 sets of logic voltages, A and B. In set A, which has been assigned to nails 1,3,7,8,9, test inputs will be driven to:

High state=+4.5 V
Low state = 0.0 V

and outputs will be sensed for:

High threshold= 2.4 V
Low threshold = 0.8 V



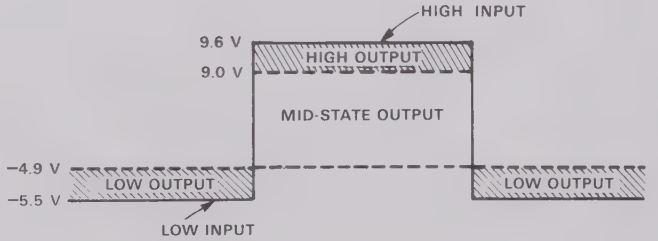
As shown in the diagram, a sensed voltage less than 0.8 V will be considered a low output, a voltage greater than 2.4 V will be considered a high output, and any voltage between these two thresholds will be considered in mid-state.

In set B, which has been assigned to nails 2,4,5,6, test inputs will be driven to:

High state=+9.6 V
Low state =-5.5 V

and outputs will be sensed for

High threshold=+9.0 V
Low threshold =-4.9 V



Once the voltage levels have been established and assigned to individual test nails, each driver/sensor is controlled by SET LGC and TEST LGC statements.

The TEST LGC command is the 'workhorse' of digital testing. It controls the input test nails by forcing the drivers to specified states and checks the expected results by monitoring the sensor outputs. The state of a nail, it is obvious, can be high or low; but for testing purposes it can also be undriven (disconnected) or 'don't care' (ignored). Before it performs a test operation, a nail must be enabled by connecting it for driving or sensing. Thus, nail states are specified in the test language as

For Driving IC Inputs

Not driven	ID	(Input Disconnect)
Driven	IC	(Input Connect)
Drive high	IH	(Input High)
Drive low	IL	(Input Low)

For Sensing IC Outputs

Not sensed	OI	(Output Ignore)
Sensed	OS	(Output Sense)
Sense for high	OH	(Output High)
Sense for low	OL	(Output Low)

Using the simple NAND gate shown in Figure 4-22 as an example, the statement

```
TEST LGC IC(21,22) IH(21,22) OS(23) OL(23);
```

controls the driver/sensors. The drivers attached to input nails 21 and 22 are connected and driven high while the sensor attached to output nail 23 is enabled and looking for a low.

Following the ASSIGN LGC statements, all drivers are disconnected from the UUT and all sensor outputs are ignored. The first time they are connected in the burst, drivers and sensors default to their low states, IL and OL, until explicitly set to other states. The TEST LGC statement above first connects (enables) the 2 drivers wired to input nails 21 and 22, forces these driver outputs to a high state, then enables the sensor connected to output nail 23. All nail states are incremental, that is, whenever a nail is placed in a given state, it remains in that state until a subsequent test step changes it to another state or until the burst ends. If a burst begins with the statement shown below,

```
TEST LGC IC(21,22) IH(21,22) OS(23);
```

the output nail 23 will be sensed for a low, the default state it assumed following the ASSIGN LGC statement, even though the term OL(23) does not appear in the statement. Likewise, the sensors attached to input nails 21 and 22 are ignored (state OI) and the driver attached to output nail 23 is off (state ID). Similarly, the statement

```
TEST LGC;
```

or simply

```
;
```

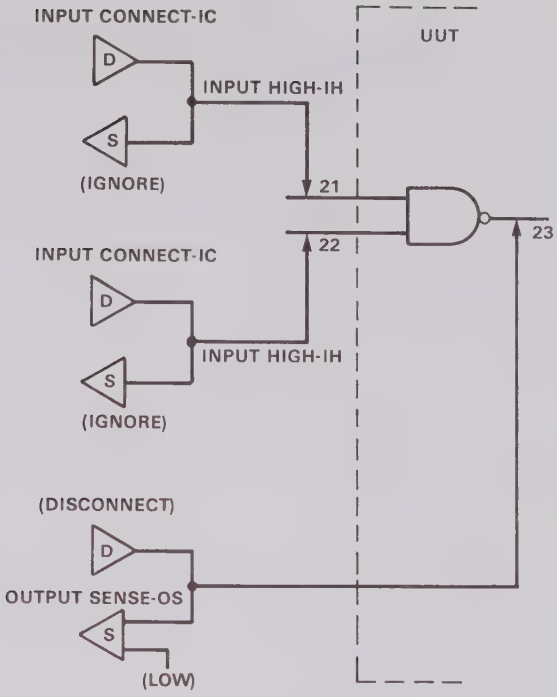


Figure 4-22. Typical Driver/Sensor Connections to UUT.

is not an empty test statement, as it would be in many programming languages. It is, rather, a repetition of the preceding statement.

To test fully the NAND gate in Figure 4-22, 4 consecutive TEST LGC statements would be used to force input nails 21 and 22 to all 4 possible input combinations and to test nail 23 for the expected results. The columns at the right show the nail state context -- the test states of all the nails -- at each step in the test. The first, 'empty' step is given simply to show the default initial states. The new states caused by each test statement are underlined.

Test Program	State Context		
	Input nails		Output nail
	21	22	23
BURST;			
;	---ID,IL,OI,OL--- /* ALL NAILS */		
IC(21,22) IH(21,22) OS(23);	IC IH	IC IH	OS OL
IL(22) OH(23);	IC IH	IC IL	OS OH
IL(21) IH(22);	IC IL	IC IH	OS OH
ID(21) IL(22) OI(23);	ID IL	IC IL	OI OH
;	ID IL	IC IL	OI OH
END BURST;			

NOTE

Since TEST LGC is the default digital statement, the keywords TEST LGC can be omitted for simplicity in writing programs. Similarly, SET LGC (described later) can be replaced by a $ symbol.

The SET LGC statement can be used to set up test inputs without simultaneously testing the resulting outputs. The format is similar to the TEST LGC statement, but all sensor outputs are ignored during the execution of a SET LGC statement. It is equivalent to a temporary OI applied to all nails.

In normal digital testing all driver outputs are checked to ensure that they reach their programmed states. In other words, the statement TEST LGC IH(1) is functionally equivalent to TEST LGC IH(1) OS(1) OH(1); if the driver is unable to drive the node to a logic high, the statement containing the IH(1) will fail because the sensor will not detect the expected logic high. (Thus,if a shorted input pin at a device cannot be driven to the desired state, the statement fails.) Occasionally it is useful to disable driver state checking, for example, when it is known that a hard-to-drive or current-sensitive node does not need to reach the programmed high or low sense voltage. An additional driver term, II(<nail-list>) (for Input Ignore) disables driver checking at all named nails. Checking is restored by IS(<nail-list>) (for Input Sense).

The pull-up and pull-down resistors attached to the output of each driver can only be controlled by a SET LGC statement. For example, as shown in Figure 4-23, the statement:

```
$ PB(25);
```

will connect a 3.3K pull-up resistor between test nail 25 and the high driver reference voltage (i.e., the VIH value defined by the ASSIGN LGC statement),

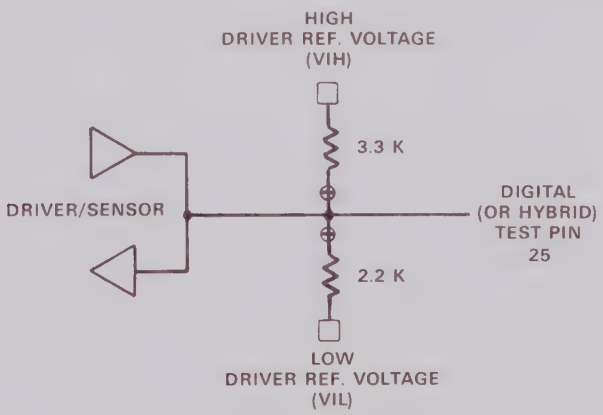


Figure 4-23. Terminating Resistors on a Digital Nail.

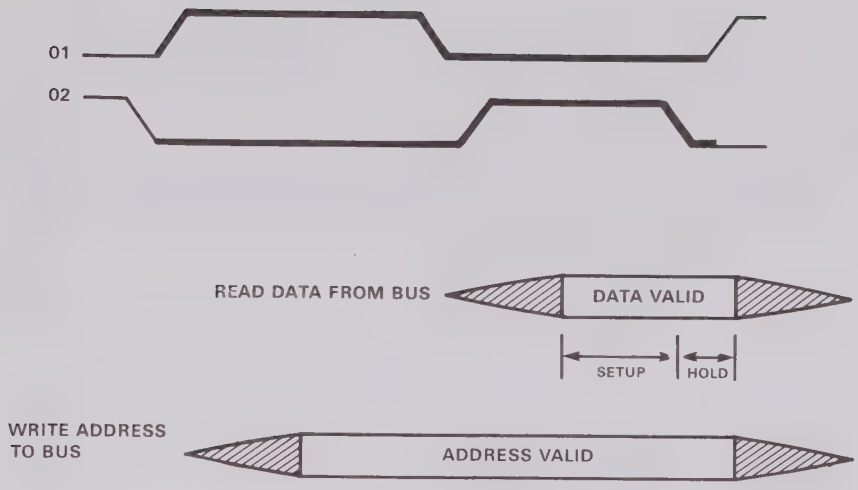
and a 2.2K pull-down resistor to the low driver reference voltage (i.e., the VIL value defined by the ASSIGN LGC statement).

An example of the use of these terminating resistors for testing tri-state devices is given in the SET LGC description in Section 6.

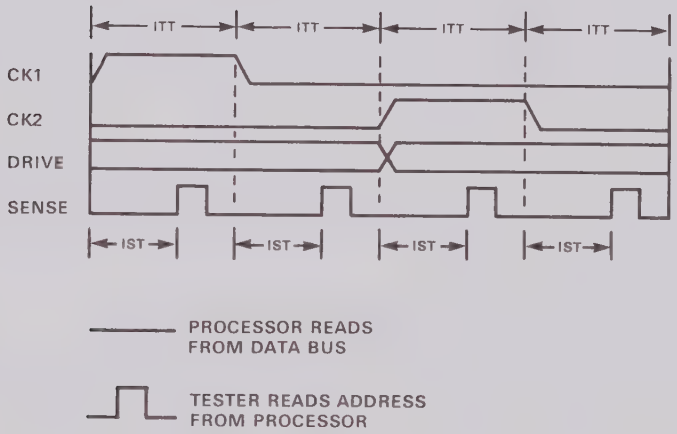
The CLEAR LGC statement resets all driver/sensors to their initial state (disconnected, ignored and low).

4.3.2 Testing Clocked Devices

Many larger IC devices have a clock pin that must be driven with a pattern of alternating highs and lows. A test for such a device therefore often contains 2 test vectors for each cycle of the clock, with alternating TEST LGC IH and IL statements being applied to the device's clock nail. Consider the



a)



b)

Figure 4-24. Test Timing for MC6800 Microprocessor.
a) Read Cycle Timing. b) Corresponding Test Timing.

timing diagram for the MC6800 microprocessor, shown in Figure 4-24. The 227X digital test library contains a model test for testing this device on the standard digital test hardware, using the testing strategy described here. To supply the 2-phase clock shown in the diagram, the library test procedure uses 2 driver/sensor nails, connected to the 2 clock nails, identified as nails 101 (phase 1) and 102 (phase 2). The waveforms provided by the driver/sensor nails do not correspond closely to those shown in the figure, but they are adequate for the test. For each clock cycle, 4 test steps are required, as shown below.

```
        /* 101 = Clock nail phase 1; 102 = phase 2 */
IH(101);                /* Send a pulse to phase 1 clock pin */
IL(101);

IH(102);                /* Send a pulse to phase 2 clock pin */
IL(102);
```

The MC6800 timing diagram reveals that, for a read operation, the microprocessor should place an address on its address nails during the phase 1 clock. To test the execution of a read (or load) instruction, the test system must therefore sense these pins during the first clock phase -- while phase 1 is still high. The test assigns 16 test nails, H1 to H16, to the microprocessor address pins, A0 to A15. Suppose the processor is expected (on the basis of some previous test steps) to read from the address FF00 (hexadecimal). Statements like these will then be used.

```
BURST ITT=4U IST=2U; /* Define test step and drive-sense timing*/

                     /* Drive phase 1 high and sense addr FF00 */
IH(101) OH(16,15,14,13,12,11,10,9) OL(8,7,6,5,4,3,2,1);

IL(101);             /* Drive phase 1 low; address still present */
```

Remember that each test statement (ending with a semicolon) consists of 2 operations, a drive pulse and a subsequent sense strobe. Therefore, the first statement above will drive the phase 1 nail (101) high, wait an interval (defined by the BURST parameter IST), and then sample the address lines H16 to H1. If the test system senses all highs on the top 8 lines and all lows on the lower 8 lines (i.e., the address FF00), then the test step passes. The next step drives the phase 1 line to a low.

On the phase 2 clock, the microprocessor expects to receive some data from the addressed device -- which the test system will usually have to supply, since all other ICs on the bus are disabled during the test. Statements to do this might be

```
                     /* Drive phase 2 high; supply data */
IH(102) IH( <data nails high> ) IL( <data nails low> );

IL(102);             /* Drive phase 2 low; uP latches data in */
```

4.4 SpeedPlus TEST DEVICES

The SpeedPlus option provides hardware and software facilities meant to simplify the testing of microprocessors and other VLSI devices. To meet the minimum operation rates required by many VLSI components, these facilities allow testing at rates up to 5 million steps per second. They also take advantage of the fact that many VLSI devices have similar fundamental structures and therefore usually require similarly structured tests. By moving some of the test functions, such as clocking and machine cycle synchronization, to special dedicated hardware, the SpeedPlus subsystem carries out most VLSI tests in a simpler fashion than would be possible with general-purpose digital test hardware. To use these functions effecti ely, the test programmer requires some understanding of how the SpeedPlus hardw re operates.

4.4.1 SpeedPlus Subsystem Organization

The hardware modules supplied with the SpeedPlus subsystem are connected to special test nails in the test fixture connector (receiver). Up to 864 (2272) or 320 (2271) SpeedPlus test nail connections can be purchased for high speed digital tests, depending on the number of driver/sensor boards supplied with the subsystem. Like the standard 2271 and 2272 test nails, all of these additional nails are provided with relay connections to the analog measurement

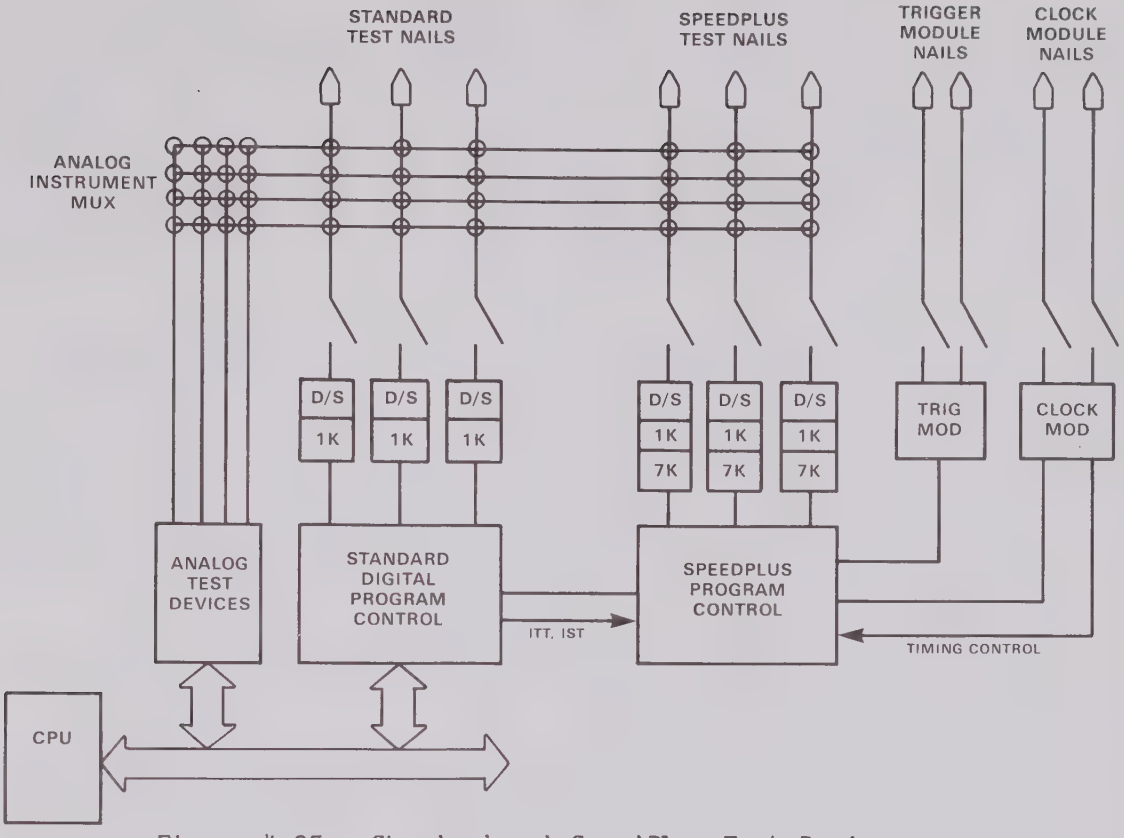


Figure 4-25. Standard and SpeedPlus Test Devices.

bus and can be used for analog tests as well. Such nails, with digital and analog capabilities, are sometimes called hybrid nails. In addition, the SpeedPlus subsystem also supplies

16 (2272) or 14 (2271)	clock signal nails
8 (both systems)	clock synchronization nails
28 (2272) or 24 (2271)	event trigger nails

These nails will be referred to collectively as the SpeedPlus timing nails; the hybrid driver/sensor nails will simply be called test nails.

As explained in para 4.3, the test system's central processor initiates a standard test by loading test vectors into a high-speed memory bank attached to the standard driver/sensors. The standard driver/sensor controller then steps through these vectors at the rate defined by the burst parameter ITT (Inter-Test Time), applying drive pulses to the test nails at each ITT time, and sampling the sense nails shortly after each drive time (with a delay defined by IST -- Inter-Strobe Time). In the SpeedPlus subsystem, each driver/sensor is provided with 2 banks of memory for test vectors, as shown in Figure 4-25. The test vectors for SpeedPlus tests are stored in the larger (7K) of these banks, and vectors for standard tests can be stored in the other (1K), so that both standard and special, high-speed tests can be conducted at nails connected to the SpeedPlus hardware. When a BURST; statement appears in a test program, it starts a digital test in the standard controller. If the burst contains test statements that refer to nails in the SpeedPlus test system, the standard controller sends a message to the SpeedPlus controller to place it in a 'slave' mode. In this mode, the SpeedPlus test nails emulate the system's standard

hybrid test nails and execute all test statements just as if they were part of the standard driver/sensor system. This allows the programmer to write standard digital test statements using both the standard test nails and any of the additional test nails in the SpeedPlus subsystem. The SpeedPlus test nails also have scanner connections to the analog measurement bus, and can be programmed to carry out standard analog (non-burst) measurements. Only the SpeedPlus timing (clock and trigger) nails cannot be used in 'slave' mode or for analog testing (except by connecting them to unused driver/sensors with the DECLARE NAILBIND statement).

Whenever the statement FAST; occurs within a burst, the standard controller places the SpeedPlus controller in its 'native' mode and passes program control to it. During the FAST section the SpeedPlus controller executes the SpeedPlus test steps independently of the standard controller, usually at a much higher test rate. It continues in this mode until the statement END FAST; appears in the burst. Since the standard nails cannot execute SpeedPlus test steps, test statements in the FAST section of a burst can refer to only those test nails that belong to the SpeedPlus subsystem. During the FAST section, the standard test nails remain in the states they had when the FAST section was entered.

Tests executed in the FAST mode make use of the SpeedPlus subsystem's timing system to furnish test step timing, clock signals, and test synchronization. In the 'slave' mode, the SpeedPlus test controller takes its timing signals from the standard ITT and IST signals that control the standard driver/sensors. In a FAST section, however, timing for the SpeedPlus driver/sensors is provided by a programmable clock module. This module uses a matrix of high-speed memory cells, programmed with highs and lows corresponding to the waveforms of the desired drive and sense strobes, as well as other control signals, that are read out at a rate as high as 40 MHz to give a maximum timing resolution of 25 ns. The clock module uses this same matrix to furnish up to 4 distinct, digitally synthesized clock waveforms to the UUT. The clock signals are therefore coordinated with the test steps, but they are programmed in an independent way, so that clock edges and driver pulses do not have to fall at the same times.

A SpeedPlus test can be synchronized to the UUT at 2 levels. At the first level, the test step rate is synchronized to a UUT clock; where an on-board UUT clock cannot be disabled, this permits the tester to drive and sense in synchronism with the clock signal present on the board. (Test debugging is, however, likely to be more difficult at a high clock rate.) The second level of synchronization is more complex; it allows 'stop-and-go' coordination of the test with particular UUT events, such as read/write cycles, resets, or interrupts. The execution of each test step in a FAST section can be made contingent on the states of up to 7 trigger nails. As shown in Figure 4-25, a trigger module together with the clock controls test step execution in the SpeedPlus 'native' mode. The trigger module compares the states of specified trigger nails to a trigger condition defined in each test step, and if a match is found, it permits the step to begin. Otherwise, it blocks the step until the trigger condition is met. Note that test steps always begin on a driver strobe.

The timing nails are not provided with hardware for general-purpose testing. Consequently, when a digital or analog test must be performed at a timing nail, that nail must be disconnected from the timing hardware and connected, instead, to a driver/sensor. This is done as shown in Figure 4-26.

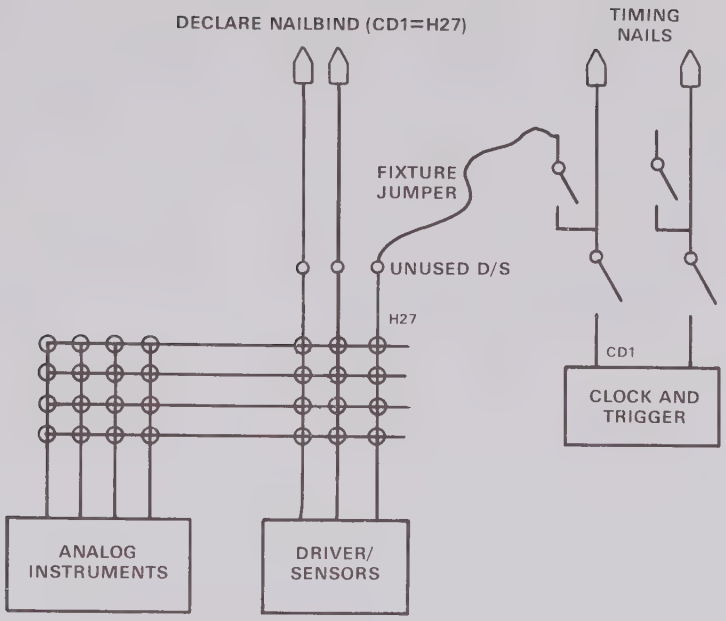


Figure 4-26. Testing at SpeedPlus Timing Nails (Nailbind).

Before the digital or analog test begins, a relay in the SpeedPlus timing module disconnects the timing nail from the timing electronics. A second relay connects the nail to a jumper wire installed in the fixture, via a special receiver connection. At the other end of the jumper, another relay to an unused test nail connection closes, so that an analog or digital test can be performed at the timing nail. (This mechanism is described fully in para 6.2, under "DECLARE NAILBIND.")

4.4.2 SpeedPlus Test Statements

TEST LGC and SET LGC statements used by the SpeedPlus subsystem are identical to those used in standard testing (IH, IL, OH, OL, PH, etc.), as described in para 4.3. Hence, test programs for the SpeedPlus subsystem have essentially the same form as standard test programs. Where standard digital test bursts are typically limited to about 480 test statements, bursts written for the SpeedPlus hardware can include as many as 7000 test statements. One standard TEST LGC parameter, OM (test for presence of a mid-state voltage), is not allowed at SpeedPlus test nails, neither in standard nor in SpeedPlus bursts. The SpeedPlus subsystem recognizes 2 additional parameters to the TEST LGC and SET LGC statements, TH and TL, which define trigger conditions in a test step. These 2 parameters are not incremental -- that is, they affect only the test statements in which they appear. A subsequent test statement has no trigger conditions unless they are explictly added.

The principal programming differences between the standard and the SpeedPlus digital testing hardware concern the SpeedPlus timing nails and the triggering mechanism. For testing VLSI integrated circuits, these nails provide dedicated hardware for functions that must otherwise be provided in the test program or in special test fixture hardware.

Clock Signals and Connections

Many VLSI devices have a clock pin that must be driven with a pattern of alternating highs and lows. Standard tests for such devices therefore usually

toggle the clock by applying alternating high and low drive levels to the clock pin, as explained in para 4.3.2. In the SpeedPlus subsystem, a clock generator module takes over this clocking function, relieving the test program of this task. In addition to 4 programmable clock waveforms, the clock module supplies signals to define the test step rate, as well as the drive, sense, and triggering times within each test step.

Some test timing features of the SpeedPlus subsytem may be easily understood by reference to the timing problems that arise in conventional VLSI testing. Consider again the timing diagram for the MC6800 microprocessor, shown in Figure 4-24. The standard test applies clock pulses that coincide with the normal drive pulses, and it has relatively little control over the shape of the clock waveforms.

Using the SpeedPlus clock module, a SpeedPlus burst is able to define the test timing in more detail than is permitted in standard digital testing. A SpeedPlus FAST section does not take the IST and ITT parameters used in standard testing. Instead, it references a SET CLOCK statement ahead of the BURST or BUSTEST containing the FAST section; this SET CLOCK sets up the drive and sense times along with a number of other timing parameters. These can include:

- up to 4 separate clock waveforms
- a synchronization waveform
- trigger sampling time(s)
- program restart time(s)
- overall cycle frequency or period
- minimum timing resolution (increment size)
- voltage levels for clock and triggering

The clock generator is, in essence, a multi-channel digital waveform synthesizer. It creates all of the timing signals necessary by dividing a basic timing cycle into a number of smaller subunits called increments. For each increment in the timing cycle, it stores the value of each clock waveform in a high-speed digital memory. It then turns this static description of the waveforms into dynamic signals by repeatedly reading out the memory locations at a high speed; at the last location the increment counter resets to the lowest increment and begins again.

Apart from the clock waveforms, the clock generator must also supply strobes to activate the drivers and sensors. Each test step in the test program will be executed by a drive and sense strobe pair, and these strobes are programmed separately from the clock waveforms in the timing memory. A sense strobe always follows a drive strobe before another drive strobe can appear. When a test step is executed, the UUT inputs will be driven to their new levels at the timing increment where the driver strobe pulse (D) appears in the timing memory, and the UUT outputs will be sampled at the increment where the sensor strobe (S) appears. Because the clock edges are not coupled to the driver strobes, as they are in standard digital testing, it is possible to drive or to sense at nearly any point in a clock cycle.

The SET CLOCK Statement

Returning to the MC6800 read cycle timing diagram as shown in Figure 4-24, consider how the various required timing signals could be programmed as a matrix of high and low values. If the full clock cycle time is divided into 12 increments, the important features of the timing diagram can be programmed quite exactly. The phase 1 and phase 2 clocks are generated by programming highs at

those parts of the cycle where the driver nail should be at the logic high voltage; the low portions of the cycle are programmed with lows.

In the test programming language, the SET CLOCK statement sets up the clock waveforms and the other timing signals required for a BURST. It is thus like any other SET statement (such as SET MUX) that sets up an instrument in preparation for a test. It precedes the first BURST that uses it and remains in effect until another SET CLOCK statement redefines the clock parameters. Normally the clock signals are applied only during subsequent bursts but, if necessary the clocks can be applied continuously (to retain data in dynamic RAMs, for example). Six parameters can be specified in SET CLOCK:

- signal-to-test-nail (multiplexer) connections
- timing cycle frequency and number of increments
- clock waveforms and synchronization edge
- driver/sensor and trigger strobes
- logic levels.
- activation time of the clock drivers

A SET CLOCK statement to generate the timing signals for the MC6800 shown in Figure 4-24a might appear as shown in Figure 4-27. This statement (note that there is only one semicolon) defines multiplexer connections, a frequency, 2 clock signals (furnished through clock nails 3 and 7), as well as driver and sense strobe times. (Other values are defined by defaults.)

```
SET CLOCK
   AT (CDA=3 : CDB=7)
   F=1.5MEG
   NINC=12
   SIGNAL (CDA=H0, L5 : CDB=H6, L11)
   DST(D8, S10)
;       /* END OF CLOCK STATEMENT */
```

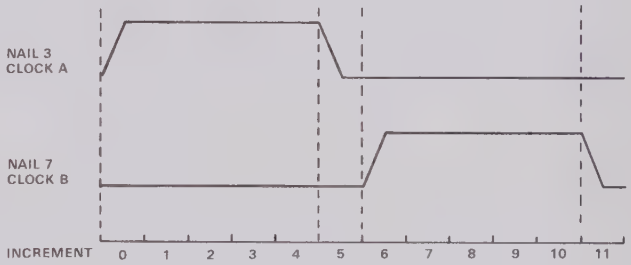


Figure 4-27. SET CLOCK Statement For 6800 Microprocessor.

Test Nail Connections

The clock module contains a nail multiplexer that, like the analog scanner, permits each of the 4 clock signals CDA, CDB, CDC, and CDD, to be sent to a number of different test nails. The synchronization input, CSYNC, from the UUT can come from any one of 8 nails. These connections are specified with an AT parameter and a nail list.

```
AT (CDA=3 : CDB=7)
```

(The receiver connections and nail numbers for the SpeedPlus nails are shown in para 6.4.4.)

Clock Frequencies and Waveforms

The next 2 parameters, F and NINC, define the timing matrix. F gives the cycle frequency -- how many times per second the timing counter sweeps through the matrix. NINC, for "number of increments," states how many intervals will be used to define the timing matrix. Each increment of the timing matrix lasts 1 / (NINC * F). On the basis of F and NINC, the optional SIGNAL parameter

describes the waveforms supplied to the UUT by the clock generator by naming the increments in which rising and falling edges occur.

```
SIGNAL (CDA=H0, L5 : CDB=H6, L11)
```

The SIGNAL parameter can also specify a waveform (not shown here) to be used in synchronizing the module to an external waveform. Because the increment numbering starts at 0, a NINC of 12 gives increments numbered 0 to 11. The waveforms presented to the 4 clock output channels are identified as CDA, CDB, CDC, and CDD. The terms after the = sign are called a transition list; they define successive low-to-high or high-to-low transitions of the waveform. "CDA=H0, L5" means: program a low-to-high transition at increment 0, and a high to low transition at increment 5 on clock A. Clock B executes a low to high transition on increment 6, and it goes low at increment 11. Within the resolution allowed by the increment size, any required digital waveform can be described.

The 2 signals defined by the clock statement of Figure 4-27 conform closely to the 2 clock phases typically used with the MC6800, shown in Figure 4-24a. At a timing cycle frequency of 1.5 MHz, each increment is only (1/12 X 1/1.5 MHz) or 55.6 nanoseconds, which allows quite exact definition of the waveforms. (The smallest increment allowed is 25 ns.) At a given timing frequency, a large value of NINC allows finer resolution of timing events within the cycle, but because the test fixture and other in-circuit testing conditions introduce unavoidable timing imprecisions, it is rarely useful to have more than 10 to 12 increments. Further constraints are imposed by the minimum times between drive, strobe, and trigger timing events.

```
SET CLOCK
   F = 1.5 MEG
   NINC = 12
   SIGNAL (CDA = H0, L2, H6, L8
           :CDB = L0, H3, L5, H9, L11)
   DST    (T1, D3, S4, ED9, S10);
```

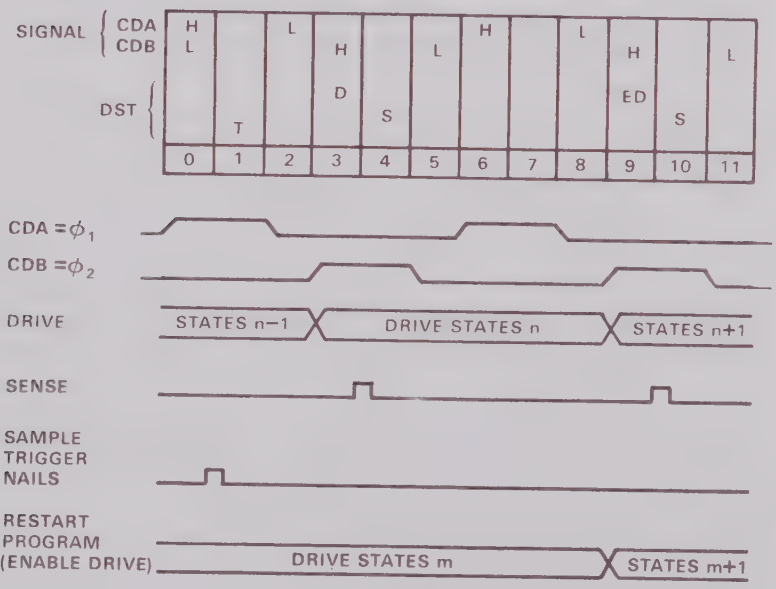


Figure 4-28. Timing Signals Generated by a SET CLOCK Statement.

Note that F defines the timing cycle frequency but does not necessarily define the resulting clock frequency. A clock timing list with 4 transitions (e.g., CDA=H0,L2,H6,L8) would yield a clock frequency of 3.0 MHz with F=1.5 MHz, as shown in Figure 4-28. As many transitions are permitted as there are increments in the matrix, and the waveform of each clock can assume nearly any shape required.

NOTE

There must be a minimum interval of 50 ns between all driver and sensor strobes.

Driver/Sensor Timing Signals

While the SIGNAL parameter defines clock waveforms sent to the test fixture, the DST parameter defines timing signals sent to the systems's internal hardware. In Figure 4-27, the parameter

DST(D8, S10)

sends signals to the driver/sensors to cause the drivers to assume their drive states on the 8th timing increment. S10 tells the sensor to sample the UUT outputs on the tenth increment. As Figure 4-28 shows, the drive and sense strobes do not have to align with any of the clock edges; they can be moved about freely with respect to the clocks, which makes SpeedPlus testing considerably more flexible than standard tests in testing clocked devices. Drive and sense times must always come in pairs, and if a DST list omits a sense strobe after a drive strobe, the default sense time (just before the next drive) will be used.

Triggering

The SpeedPlus subsystem also provides a triggering facility that allows a test step to define a set of nail states that must be reached before the step executes. Standard digital tests can wait on a condition only by entering a loop containing an EXITIF statement, such as the loop shown in Figure 4-29, used to initialize a counter that has no clear or reset input.

```
      LOOP=20
      $ IH(6);
      $ IL(6);
      END LOOP EXITIF OS(7,8,9) OH(7,8,9);
NEXT:   < a test statement >
```

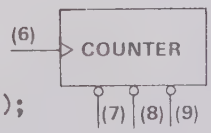


Figure 4-29. Conditional Program Control via LOOP.

These statements (explained in detail in para 6.4, "LOOP,") clock the counter until it reaches the required state, namely 1 1 1, then transfer control to the next test step. The first 2 statements, containing the pattern SET LGC IH(6); SET LGC IL(6);, serve only to clock the counter until the exit condition is reached. The EXITIF statement then 'triggers' on the desired condition, and transfers control to the next test statement. With more complex devices, however, it is conceivable that the states of other nails might need to be sampled in the same test step as the one in which the exit condition is met. Suppose, for example, a complex processor operation of unknown length has been initiated which will terminate with a write operation to memory. If the test

program tests the 'WRITE' line on each clock cycle, using a loop, like the one shown above, it will have to wait until the next test step to sample the data bus -- at which point the data may no longer be available.

Using an EXITIF loop permits a trigger condition, defined as the loop exit condition, to control the execution only of a subsequent test step. The SpeedPlus subsystem has an independent triggering mechanism, illustrated in Figure 4-30, that is somewhat similar to that of a logic analyzer. This mechanism allows a trigger condition to be specified as part of the same test step that it controls, and it thus overcomes the limitations of the conditional loop. In addition, it eliminates the extra test steps needed to define the loop and the trigger condition.

Any TEST LGC or SET LGC statement carried out on the SpeedPlus subsystem can define the state of up to 7 special SpeedPlus trigger nails as a trigger condition. Trigger conditions are specified in the same manner as other nail statements. TH(<nail list>) defines the nails that must be sensed high, and TL(<nail list>) defines the nails that must be low. Whenever at least one trigger nail is mentioned in a test statement, the test step will not execute until after the trigger condition is met. Once the trigger condition is met, the step executes at the next enable drive time (ED). For example, the statement

```
TH(7) OS(4007,4008,4009) OH(4007,4008,4009);
```

will not execute until trigger nail 7 has been found in a high state and the clock module issues ED. (Refer to para 6.4.4 for a description of SpeedPlus subsystem nail numbering.)

If more than one trigger nail is specified, the test step waits until all of the nail states appear together, that is, until a logical AND of all trigger comparisons is true.

```
TH(7,8) TL(9) OS(7,8,9) OH(4007,4008,4009)
```

This statement will not execute until trigger nails 7 and 8 sense highs and trigger nail 9 senses a low. Once the condition is met, test nails 4008, and 4009 are checked for highs.

The trigger nails are not monitored continuously. Instead, additional parameters in the DST (driver/sensor timing list) portion of the SET CLOCK statement can define the increments on which the states of the trigger nails are to be sampled and stored. If trigger conditions are specified but a trigger sampling interval is not explicitly defined, the default trigger sample time occurs just ahead (typically 60-to-100 ns) of each driver strobe in the test.

NOTE

There must be a minimum of 60 ns between a trigger sample and the enable driver (ED) time. Default trigger and drives strobes do not necessarily coincide with defined timing increments.

The DST parameter T (trigger sample) names the increment on which the trigger nails should be sampled, and the parameter ED (enable drive) defines the increment at which a successful trigger can restart the test. At trigger sample time, the states of the trigger nodes are stored, as shown in Figure 4-30. Test statements containing no trigger terms execute as soon as the next drive-sense

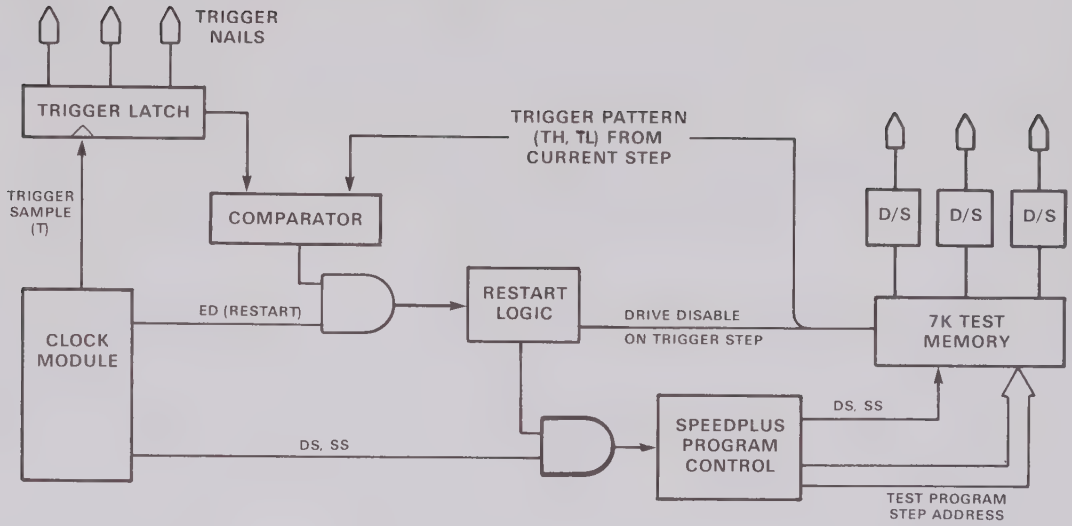


Figure 4-30. Trigger Sample and Enable Drive Logic.

pair (D, S) occurs. At the beginning of a test step that contains trigger parameters, however, currently stored trigger nail states are compared with the trigger states specified in the test step. If the 2 patterns match -- the stored nail states and the expected nail states -- the rest of the step can be carried out, starting at the next enable drive (ED) time. All D, S pairs are skipped, and the test step is suspended, until an ED occurs. (If no ED is specified in the DST list, all Ds default to EDs.) If no match is found, the clock module continues to generate clock pulse and trigger samples, but the test step does not execute. The drivers remain in the states specified by the last test step, and test nail sensors are not sampled. Two events must occur before the suspended test step can execute: at some T event, the required trigger nail states must be found, and a subsequent ED event must occur. This ED gives the drive strobe for the resumed test step. (ED can be regarded as the only kind of drive strobe recognized by a test statement containing trigger terms. Other test statements recognize both D and ED.)

Figure 4-30 shows schematically how the triggering mechanism interacts with the program control in the SpeedPlus subsystem.

4.4.3 Overall Test Event Timing

Keeping in mind the timing pattern of Figure 4-28, consider now how a short sequence of digital test statements in a FAST section might execute when the test timing is defined by that fairly complex pattern.

Recall that in a non-FAST burst section, each test statement is associated with just 2 events: a drive strobe, or time at which the drivers change state, and a sense strobe, or time at which data from sense nails is sampled. The sense strobe follows the drive strobe, and the time between them is IST; the period of the drive (and the sense) pulses is given by ITT. The test frequency is defined by 1/ITT. Normal, non-triggered test steps in a FAST section are likewise associated with 2 drive-sense timing events defined in the timing matrix; each non-triggered step is carried out by a D or ED drive strobe and a subsequent sense strobe (S).

The execution of a triggered test step is more complicated, however. A triggered test statement is associated with 3 separate timing events, occurring in the order:

- a trigger sample (defined by T)
- a restart drive, or drive enable, event (defined by ED)
- a sense strobe (defined by S)

A triggered test step ignores normal drive strobes (D events); it cannot execute until an ED event occurs. The first event in the step is a comparison of the expected trigger nail states (the TH and TL terms) with the nail states stored at the last T event -- whenever that may have been (though not within 60 ns of ED). If the 2 pattterns do not match, the step is blocked. At each subsequent ED time, however, the comparision will be repeated, and if and when a match is found, the step will execute immediately. The drivers (specified in the IH and IL terms) slew to their required states, and sensing of the specified sensor terms (OH, OL) occurs on the following S strobe. A blocked, triggered step will, of course, remain blocked until a T strobe occurs and the required trigger nail states are latched in.

The timing pattern shown in Figure 4-28 is particularly complex because it associates 2 drive-sense pairs with just 1 trigger event and just 1 enable drive event; moreover the trigger and the enable occur on different drive-sense pairs. Leaving aside the clock timing defined by SIGNAL, we have this timing definition:

```
SET CLOCK F=1.5 MEG
          NINC=12
          DST(T1, D3, S4, ED9, S10);
```

Figure 4-31 shows 4 repetitions of this clock cycle; each timing cycle contains one trigger sample event (at increment 1) and 2 drive-sense pairs (at increments 3 and 4, and at increments 9 and 10). Enable drive occurs at increment 9 (ED9). In these 4 timing cycles 8 non-triggered test steps could occur; but with triggering, it could be fewer, since any triggered step could be suspended indefinitely.

Consider now how the following test program fragment would execute, using the timing defined in Figure 4-28.

```
[no trigger]  IL(4003)  OH(4004);  /*At any drive strobe, D or ED*/
TL(2)         IH(4003)  OL(4004);  /*Only at ED, and if T2 was low*/
TH(2)         IL(4003)  OH(4004);  /*Only at ED, and if T2 was high*/
              IH(4003)  OH(4004);  /*Any drive strobe, E or ED*/
```

Figure 4-31 illustrates the effects of triggering in this test. (The test statements are shown vertically at the top of the figure. D/S nails 4003 and 4004 are shown simply as 3 and 4) Two test steps can normally execute in the course of each full timing cycle. The drive-sense timing is the same for each drive-sense pair, with the sense strobe following the drive strobe by 1 increment. Assume that the test has been synchronized with the timing cycle in such a way that the first test step coincides with the first half of the timing cycle.

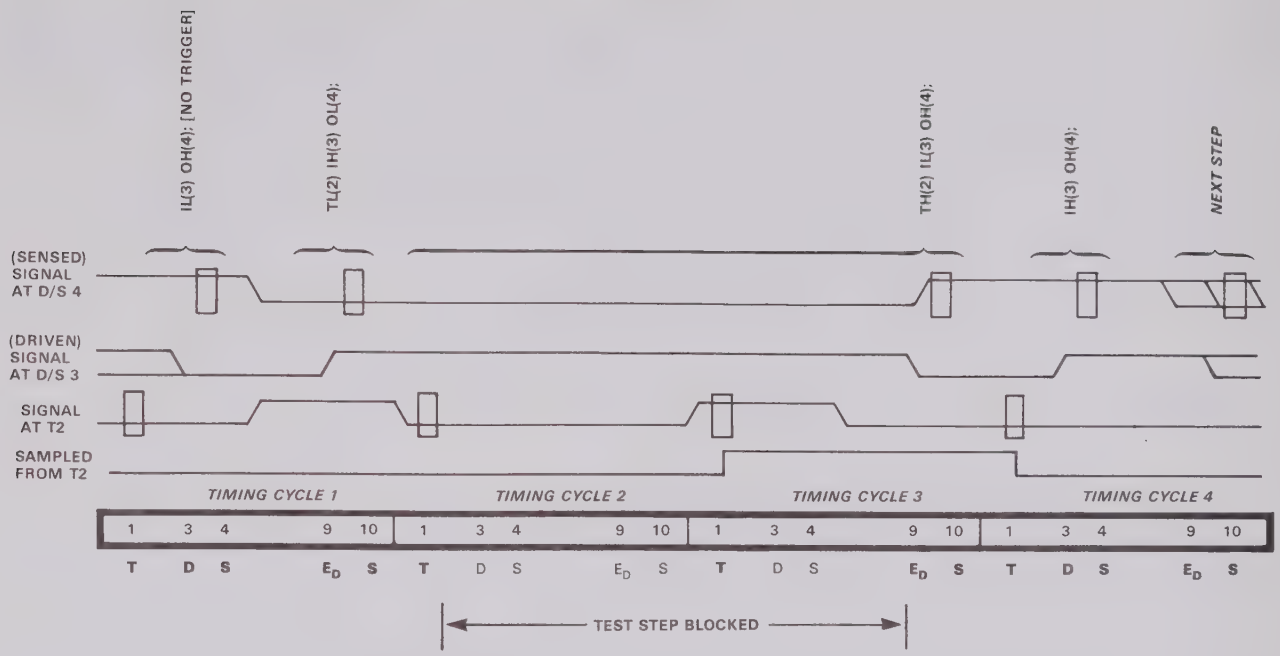


Figure 4-31. Test Execution with Trigger and SET CLOCK.

Because the first test step has no trigger condition specified, it always executes, regardless of where in the timing cycle it falls. Here it falls in the first half of the cycle; the step drives nail 4003 low at increment 3, and checks nail 4004 for a high at increment 4. Before the step executed, a trigger sample event occurred; the state of trigger nail 2 was sampled and stored as low at increment 1. The second step is a triggered step because of the test statement term TL(2); it can therefore execute only on an ED event -- which, as it turns out, occurs at this point in the timing cycle. The step specifies that trigger nail 2 should be found low; however, no trigger strobe appears with the second drive-sense pair, so the last-latched trigger nail state is examined instead. Since a low was previously sampled from nail 2, this step executes, even though trigger nail 2 is now actually high. Thus, at the ED9 strobe in timing cycle 1, test nail 4003 is driven high, and at the following S10 strobe, nail 4004 is checked for a low.

The third step specifies that the trigger nail should be high (TH(2)) before the step executes. Since the step contains a trigger condition, it can only execute at the next ED, in the second half of cycle 2. However, trigger nail 2 is sampled (in the first half of the cycle) as low. The step is therefore suspended; its next chance to execute is in the second half of cycle 3. No further test steps are executed, but clock cycles continue to occur. Altogether 3 drive-sense pairs are lost (shown in light type), which in standard digital testing would have consumed 3 test steps.

Because the trigger nail goes high at the end of cycle 2, its state is recorded as high when the next trigger sample strobe occurs in cycle 3. At the next enable drive time, therefore, the third test step drives nail 4003 low and tests nail 4004 for a high.

The fourth test step contains no trigger terms, so it can execute at the next drive strobe, which is D3 in timing cycle 4.

Clearly, the way in which the test steps defined in the test program actually execute depends on the way in which clock events are defined and on how the test steps synchronize with the clock events. As a general rule, the default clock timings should be used unless other timings are definitely required. In a case such as the one described here, the test program would need to be synchronized to the timing pattern before executing the triggered test statements. This is done most easily if no more than 1 ED time is defined within the timing list. One way to achieve synchonization is to write a statement that includes a 'dummy' trigger on a known node, such as

```
TH(VCC) ID(<all nails>) OI(<all nails>);
```

This statement can only execute on the ED event, ensuring the synchronization of all subsequent non-triggered statements.

Driver/Sensor Timing

In Figure 4-28 the clock and driver strobe values are programmed at the times when the drivers should start slewing to their required logic values. In standard digital testing it is rarely necessary to take account of fine timing details, but when tests are run at high speeds, test design requires some attention to timing detail. Because every test fixture and UUT logic family will give different effective driver slewing rates at the test nails, a certain amount of empirical measurement should precede the final implementation of a high-speed test.

As a rule of thumb, it can be assumed that clock nails will slew at around 125 V/us, and that driver test nails will slew at around 70 V/us. If very heavily loaded by a test node presenting a few ohms or less, the driver slew rate can, however, drop to as low as 10 or 20 V/us. Because nails that must be backdriven will necessarily slew slower than unloaded nails, they may need to be driven sooner than unloaded nails. At the highest test rates it may occasionally become necessary to place IH or IL commands addressing heavily loaded nails 1 or 2 test steps ahead of the steps in which they are required in order to compensate for the slewing delay.

This 'anticipated driving' requires, however, that you disable the test system's normal checking of driver states until the test step where the driver will actually reach the programmed state. In normal testing, every drive nail is checked at the IST (standard test) or S (SpeedPlus) time to see that it has in fact reached the intended drive state. If, because of reduced slew rate, the nail cannot attain the required level by the sense time, the test step will fail. (Refer to para 4.3.1 for further discussion of this mechanism.) Suppose, for example, that test nail 4001 must drive a highly capacitive node at a high test rate. It may take 2 test step times for the driver to charge the node to the programmed logic level. In this case, program steps like the following would be required.

```
IC(4001) IH(4001) II(4001);
< next TEST or SET LGC; >
IS(4001);
```

The first of these steps connects a driver to nail 4001 and tells it to drive the nail high. Because it is known that the nail will not reach the high state during the present test step (by the time of the sensor strobe, S), driver checking must be disabled for the nail; otherwise the step will automatically fail (refer to para 4.3.1 for a discussion of II and IS). After 2 further drive

(D) times, however, the nail should have slewed to its programmed state. To ensure that it now is in the correct state, the IS(4001) term reenables driver checking in the third step. If the nail is not at the high logic level by this time (due to a short, for example), this third test step will then fail.

NOTE

In general, timing resolution of less than 100 ns will require more attention to fixturing and timing definition than is likely to be worthwhile. The problems arising from drive nail skew should be minimized by running a test no faster than is required for proper operation of the UUT. In-circuit testing should not be relied on to verify timing parameters of a component.

4.4.4 Clock Synchronization.

If necessary, the SpeedPlus clock module can be synchronized to a signal on the UUT. If, for example, a UUT clock cannot be successfully disabled, it may be feasible simply to synchronize the test program to the on-board clock. If the cycle time of the clock module (the time required to go through the entire timing matrix) is specified by the programmer (in the SET CLOCK F parameter), within limits of about ±10 percent, it can be synchronized to a signal taken from the UUT. This should only be done when it is not possible to override or disable the clock signal on the UUT, or when a board-level functional test is to be run under control of the board's own clock.

Synchronization is programmed in the SET CLOCK statement by adding a synchronization nail to the AT parameter and a CSYNC term to the SIGNAL parameter, for example,

```
AT (CDA=3 : CDB=7 : CSYNC=1)
SIGNAL (CDA=H0, L5 : CDB=H6, L11 : CSYNC=H0, L5)
```

Using this SIGNAL statement, the clock module creates a third clock waveform, in addition to CDA and CDB. During a test it adjusts its internal timing so that the waveform described by the CSYNC transition list (H0, L5) coincides with the signal fed to it from synchronziation nail 1. The CSYNC waveform should contain one or more clock cycles in the required phase relation to the other timing signals. Otherwise, the DST transition list may not be properly aligned with the DUT clock. In this example, clock A (which may not be used) will follow the on-board clock, and clock B will follow it at a displacement of 6 increments. A driver strobe defined as D0 will coincide with the rising edge of the on-board clock.

A test that requires synchronizing the clock module to the UUT needs to check that synchronization has been achieved before proceeding. It does this by executing a TEST CLOCK statement before entering the burst.

```
TEST CLOCK
   MAXTIME=<seconds>
   FAIL=<fail bit number>
   [ <error processing label> ]
;
```

If synchronization cannot be achieved within the time given as MAXTIME, a fail bit can be set or a branch can be taken to an error handling routine.

Synchronization can also be tested within a FAST section of a burst by executing a conditional branch of the form

GOTO <destination label> NOPHASELOCK;

If the clock module fails to synchronize before a burst begins, the error message "NO SYNCHRONIZATION ACHIEVED" is written to the MESFILE; the burst writes "SYNCHRONIZATION LOST" if sychronization is lost at any time during the burst.

Precise definitions of the SpeedPlus statements appear in para 6.4.4 and 6.4.5 of this manual; suggestions for testing specific categories of VLSI devices appear in para 6.2 of the Test Library Programming manual.

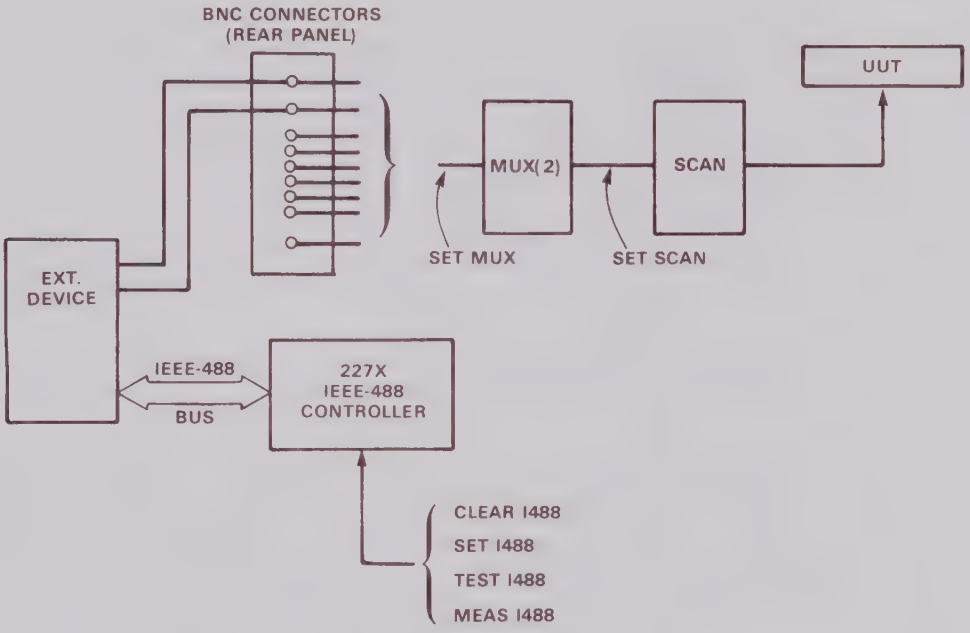
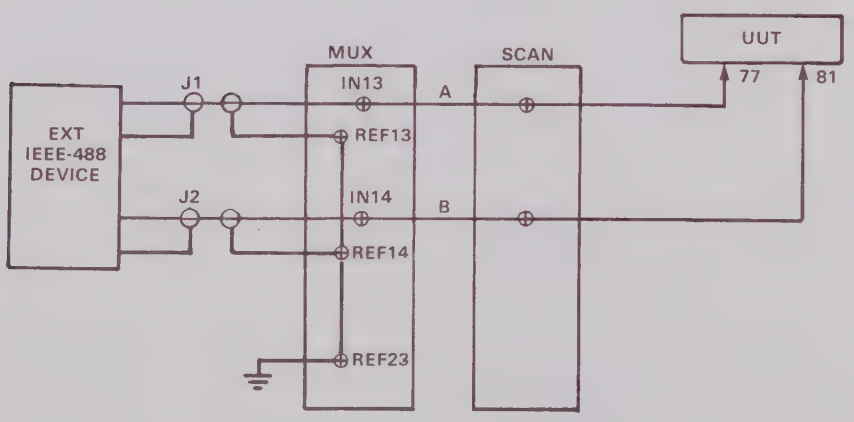


Figure 4-32. IEEE-488 Bus Option.



SET MUX AT (CHA=13: CHB=14: REF=13, 14, 23);
SET SCAN AT (CHA=77: CHB=81);

Figure 4-33. Typical IEEE-488 Bus Connections.

4.5 EXTERNAL TEST DEVICES (IEEE-488 BUS OPTION)

The system can interface with up to 9 external devices conforming to IEEE standard 488-1978. These devices can be connected to the UUT and remotely operated via test language statements. This IEEE-488 option extends the capabilities of the system by allowing the user access to additional test instruments, not available as part of the system itself.

Figure 4-32 shows a simplified diagram of a typical IEEE-488 device hook-up to the system.

The source and/or measure lines from the external IEEE-488 device plug into BNC connectors on the rear panel of the processor bay. These connectors are wired internally to input nails 13-24 of optional MUX(2). SET MUX and SET SCAN statements can then be used to complete the connections from the MUX(2) inputs to the appropriate UUT pins.

Figure 4-33 shows how a typical device would be connected to UUT pins at nails 77 and 81.

The SET MUX statement,

```
SET MUX AT (CHA=13: CHB=14: REF=13,14,23);
```

connects the device lines to MUX channels A and B and connects system ground to the device ground. The SET SCAN statement,

```
SET SCAN AT (CHA=77: CHB=81);
```

then connects MUX channels A and B to the UUT via test nails 77 and 81, respectively.

As shown in Figure 4-34, the operation of the device itself is controlled by test language statements via the IEEE-488 Controller option in the system. The controller converts the statements into appropriate control and data signals to the external device. Table 4-1 briefly summarizes the general handshaking that takes place between the system and the IEEE-488 device, under program control.

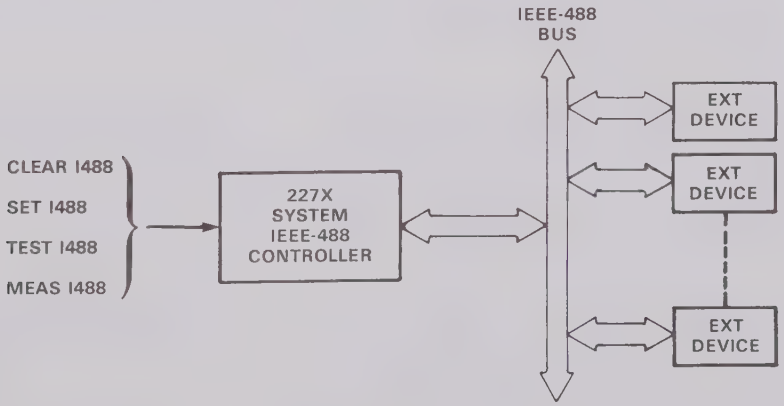


Figure 4-34. IEEE-488 Bus/System Interfacing.

The specific handshaking sequence between the I488 controller and a particular device is a function of that device. The I488 description in Section 6 of this manual contains a detailed programming example for interfacing the Fluke [1] 1953 Counter.

Table 4-1
IEEE-488 HANDSHAKING COMMANDS

Command	Action
CLEAR I488;	Activates the Interface Clear (IFC) line to all devices on the IEEE-488 bus.
TEST I488 MAXTIME=secs line [error-statements];	Checks the line specified, SRQ or EOI (default is SRQ) for a timeout defined by MAXTIME=secs. If the timeout expires before the line gives an affirmative response, the error-statements are executed.
SET I488 function [error-statements];	Broadcasts a bus command specified by function to the IEEE-488 data bus. If the command is not acknowledged within the timeout, the error-statements are executed.
READ ID=I488 address LENGTH=nchars 'message' [error-statements];	If address is specified, places the addressed instrument in talk mode. Reads until nchars characters are received into string and float variables specified in the parameter message. If the timeout expires for any character, the error-statements are executed.
WRITE ID=I488 address 'message';	Places the addressed instrument in listen mode, then sends it commands and character strings defined in message.

The commands CLEAR I488, TEST I488, and SET I488 are used for general bus control; READ ID=I488 and WRITE ID=488 are used to exchange measurement commands and data with instruments on the I-488 bus.

IEEE-488 instruments communicate by exchanging ASCII characters and symbols. The READ and WRITE allow information to be sent or received from the IEEE-488 instruments via string or float variables defined elsewhere in the test program. The variables can be intermixed in any order. Suppose, for example, that an instrument sends first a character string giving the value of a measurement, then a space, then a second character string to identify the measurement units. The first string can be read directly into a float variable, and the second can be read into a string variable. Using string manipulation commands provided in the test language, the float can then be normalized to another unit.

[1] John Fluke Mfg. Co., Inc., Mountlake Terrace, WA.

4.6 UUT POWER SUPPLIES

The system has several power-supply options that can be used to apply power to the UUT during in-circuit digital testing (or functional testing) or to power other auxiliary test circuits employed by the user (such as relays, LEDs, etc). The capabilities of these various supplies are summarized in Table 4-2.

A more detailed description of these supplies can be found in Section 6 of this manual and also in the Test Set Preparation manual.

As shown in Figure 4-35, the output of each supply is available at the receiver connector. When used to power the UUT, each supply output should be connected to the UUT through a relay. Each HCS has its own internal relays for this purpose. Optional relay boxes controlled by relay drivers on the PIO board are available for the TPS and UPS. Using relays at the output of each power supply makes it easy for the programmer to attach or detach the power supply from the UUT, as the test program demands. Refer to the Test Fixture Kit manual for your system for further information.

Table 4-2
UUT POWER SUPPLIES

Fixed Supplies	
+5 V, 3 A	Standard on all systems. Voltages are not programmable, but on a 2275 system, the fixed supply can be turned on and off, and connected to the UUT under program control.
±15 V, 0.75 A	
Triple Power Supplies (TPS)	
(2270,2271,2272)	System may contain up to 2 optional TPSs, one in slots 1-3 and one in slots 4-6 (if system has no UPSs). Each of the 3 sections of a TPS has 3 available voltage levels (VA, VB, VC) preset by hardware (resistor plugs) and selectable by the program. This type of TPS is not used on the 2275.
(1) 0.5 - 7.5 V, 5 A	
(2) 0.5 - 20 V, 0.5 A	
(3) 0.5 - 20 V, 1.5 A	
Triple Power Supplies (TPS II)	
(2275)	System can contain 1 optional TPSII that is located on the bottom shelf of the CPU bay. Each of the 3 sections of a TPS II has 3 available voltage levels (VA, VB, VC) preset by hardware (resistor plugs) and selectable by the test program.
(1) 0.5 - 7.5 V, 7.5 A	
(2) 0.5 - 20 V, 3 A	
(3) 0.5 - 20 V, 3 A	
Universal Power Supplies (UPS)	
0 - 10 V, 4 A (1 slot)	System may have optional UPSs in slots 4-6. Outputs are programmable through the whole voltage range.
0 - 20 V, 2 A (1 slot)	
0 - 40 V, 1 A (1 slot)	
0 - 80 V, 0.5 A (2 slots)	
High-Current Supplies (HCS)	
0 - 7 V, 17 A	Up to 4 supplies of either or both type available for UUTs requiring large current capabilities. Each HCS is programmable through its whole voltage range via an optional dc source unit. Bipolar output selectable by changing jumpers. Not available on 2275.
0 - 5 V, 35 A	
(2270 optional 1 HCS)	
(2271 optional 2 HCSs)	
(2272 optional 4 HCSs)	

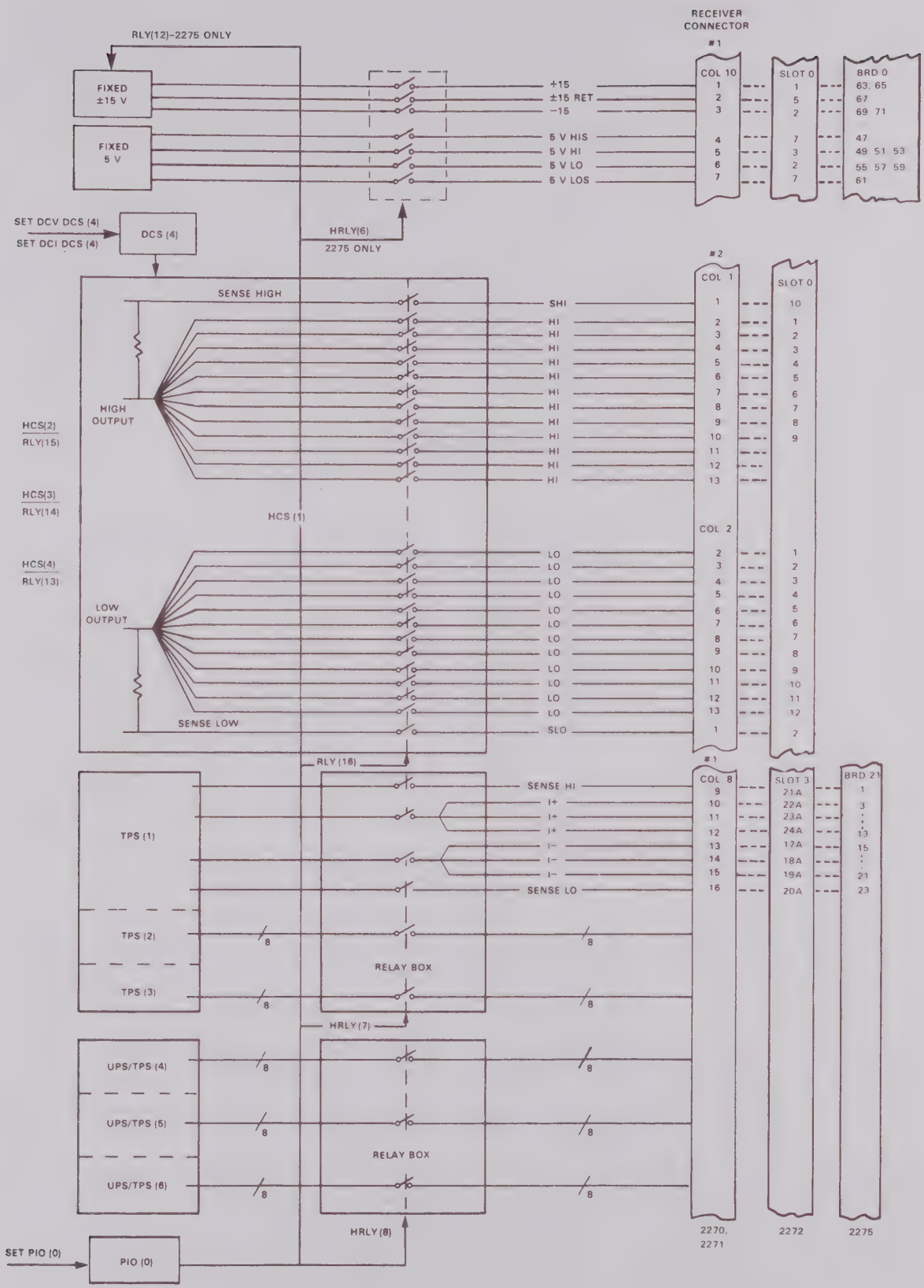


Figure 4-35. UUT Power Supplies.

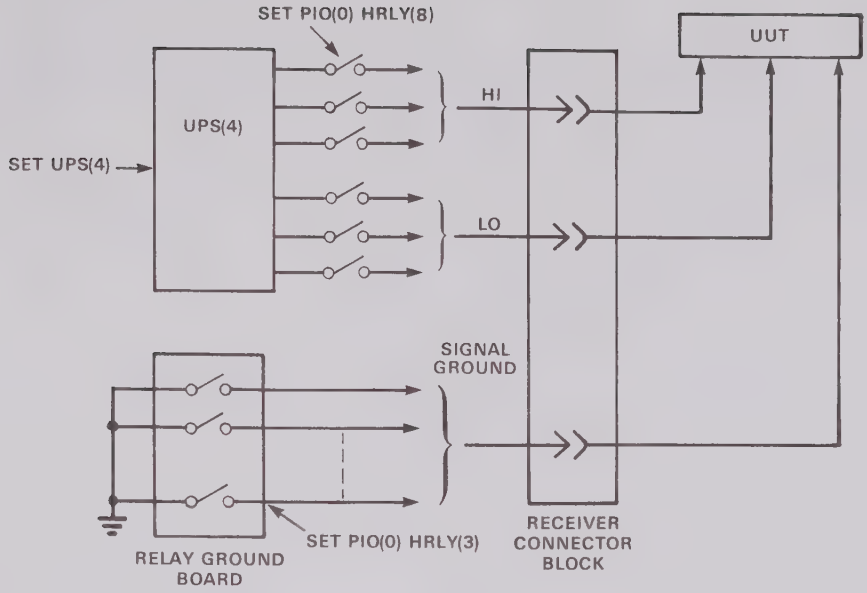


Figure 4-36. Typical UPS Connections.

For in-circuit digital testing or functional testing, both system power and signal ground must be applied to the UUT. To simplify connecting and disconnecting ground from the UUT, the system contains special ground relays that connect system ground to the bottom row of connector nails on the 2270/2271 test fixture (refer to para 2.4 in the System Operation manual) or to fixture ground nails on the 2272. By wiring the UUT ground node to these fixture nails, program-controlled relay driver HRLY(3) on the PIO(0) module can be used to connect and disconnect system ground from UUT ground. (See PIO description in para 4.6.)

Like most of the other system devices, the UUT power supplies are also controlled by SET, CLEAR, TEST and MEAS statements. The specific instruction format for each supply is described in Section 6, but a few sample statements are presented here to illustrate typical applications.

Figure 4-36 shows typical power and ground connections to a UUT (using the UPS(4) supply). The following statement sequence would be used to connect power and ground and to turn the UPS on.

```
SET PIO(0) HRLY(CLOSE 3);
```

activates the Relay Ground board via the system PIO board, connecting system ground to the UUT.

```
SET PIO(0) HRLY(CLOSE 8);
```

closes the relays in the optional relay box attached to the UPS(4) outputs, connecting the UPS to the UUT.

```
SET DCV UPS(4) V=5 I=.5 DLY=20M [WRITE 'UPS(4) DID NOT SETTLE';];
```

sets the output voltage to +5 V, the current limit to 0.5 A, and waits 20 ms for the supply output to settle. If the supply does not settle within that time, an error message is displayed.

The TEST and MEAS statements, like those associated with the analog test devices, either test or measure the supply output against specified limits; e.g.,

```
TEST UPS(4) V=5 TOL=10 [CALL ERR];
```

checks that the 5V output is within ±10% and calls subroutine ERR if it is not;

or

```
MEAS UPS(4) INTO VVAL;
```

stores the measured output value in variable VVAL.

A special MEAS DEVTYPE statement allows the programmer to interrogate a specified slot to determine what type of supply it contains. For example,

```
MEAS UPS(4) DEVTYPE INTO PS4;
```

will place a 0 in variable PS4 if slot 4 has no UPS; otherwise, it will place a 10, 20, 40 or 80 in that variable to show the type of UPS located in that slot.

The CLEAR statement resets the designated power-supply output to 0 V.

4.7 UTILITY DEVICES

The system has 3 utility devices that are not directly associated with the actual testing but which can help the user check out and set-up the test hardware and can help the programmer implement the test program. Two of these devices, the Parallel I/O (PIO) module (already introduced in the previous paragraph on power supplies) and the Self-Test Module (STM), are hardware units controlled by the test language. The third device, an arithmetic (ARITH) test software module, allows the program to test quantities not directly measured by the system and to react to an arithmetic test failure in the same way that it would to a hardware test failure.

PIO

Each Parallel I/O module contains a number of independently controlled relay drivers, TTL drivers, TTL sensors and latches. A system contains 2 PIO modules: (1) PIO(0) is reserved for system use (i.e., activating the test fixture vacuum, monitoring the test fixture switch, connecting and disconnecting system ground to the UUT, controlling power supply relays); (2) PIO(1) is for the customer's general use and can be employed to control auxiliary test circuits, relays, indicators, etc. Besides being able to control, individually, the output of each relay or TTL driver, the user can also read back the existing state of each driver, sensor and latch on the PIO board.

The SET PIO statement is used to control the driver outputs; e.g.,

```
SET PIO(1) HRLY(CLOSE 1);
```

will activate high-voltage relay driver HRLY(1) on the user PIO board and leave all other HRLY outputs unchanged. (See Figure 4-37.)

Special keywords are allowed to indicate how the drivers will be activated or deactivated (e.g., OPEN, CLOSE, JAM, MBB-make before break, BBM-break before make). Also, to simplify the construct of PIO statements, the # (all-but) symbol is allowed. For example,

```
SET PIO RLY(OPEN #5);
```

will deactivate (open) 15 of the 16 RLY drivers but will leave RLY 5 unchanged. See PIO description in Section 6 for more details.

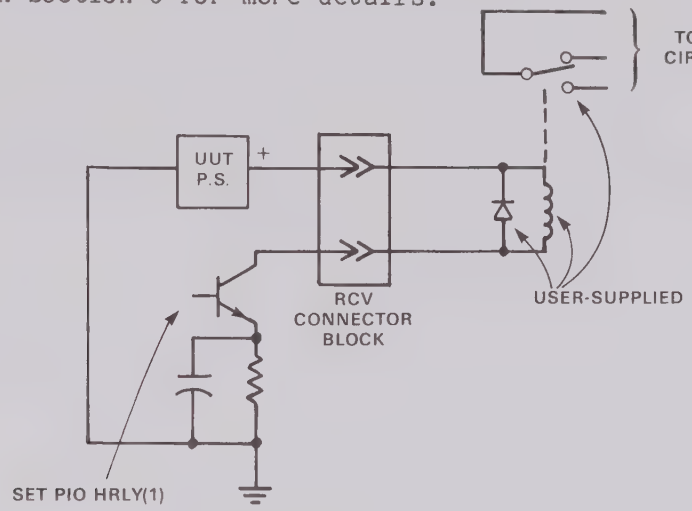


Figure 4-37. Typical Relay Driver Connections.

The MEAS PIO statement reads back the state of all devices in the specified group. For example,

```
MEAS PIO(1) ITTL INTO SENS;
```

will read into variable SENS a 16-bit string reflecting the present state of the 16 TTL sensor inputs (indicated by the keyword ITTL) on the PIO(1) board. Individual bits within this bit string can then be investigated to determine the state of the corresponding device. Assuming the following bit string is read in:

```
          0 1 1 0 0 0 1 0 0 0 1 0 0 0 1 1
          ¦
bit 16----
```

The following IF statement could then be used to test the state of any sensor. For example, bit 6 can be tested as follows:

```
IF SENS(6) THEN BRANCH S6;
```

That is, if bit 6 is a one, then the program branches to location S6.

A CLEAR PIO statement resets all drivers and latches on the PIO board.

STM

The Self-Test Module contains circuit components (i.e., resistors, capacitors, inductors, diode, voltage source) that can be connected to the 4 MUX channels (CHA-CHD) via program-controlled relays. By closing the appropriate relays, various system sources and measurement devices can be connected to known components on the STM. These components can then be measured, the results evaluated and system operation verified. The Operation Verification Programs (OVPs) available with the system use this STM module for checking system operation.

The STM can also drive 14 LEDs and monitor 12 user-operated sense switches, all located on the STM module. In addition, the 4 MUX channels can be gated to external connectors, EXTA, B, C, D, located on the STM board.

The LEDs and relay drivers are controlled by SET STM statements and the state of the sense switches and drivers can be read back by MEAS STM statements.

ARITH

The TEST ARITH statement gives the programmer recourse to all the system instrumentation failure options whenever the tested arithmetic value fails to fall within the specified limits. For example,

```
TEST ARITH BETA LO=10 FSUB=FAILSUB FAIL(5);
```

will check the existing value of the variable BETA. If the value is less than 10, fail bit 5 will be set and the designated failure reporting subroutine FAILSUB will be automatically called.

4.8 INSTRUMENTATION STATEMENT FORMAT

The general format of an instrumentation statement is as follows:

```
command function device (unit) parameters;
```

where:

command — keyword that defines the action to be taken (i.e., SET, TEST, MEAS, CLEAR, or ASSIGN). Always appears first in the statement.

function — keyword that specifies the function to be supplied, tested, etc. The functions are device-dependent and may specify such items as DCV, CS, Z, etc. Functions applicable to a particular device are covered in that device description.

device — keyword that specifies the instrument being used in this statement. They include:

ACZ	- AC impedance measurement unit
ARITH	- Arithmetic test
DCM	- DC measurement unit
DCS	- DC source (also used for HCS)
I488	- IEEE (or IEC) bus
LGC	- Digital driver/sensors
MUX	- Instrument multiplexer
OPENS	- Open circuit detector
PIO	- Parallel I/O (Relay Driver board)
R	- Ohmmeter
SCAN	- Nail scanner
SHORTS	- Short detector
STM	- Self-test module
TPS	- Triple power supply
UPS	- Universal power supply

(unit) identifies a particular unit if the system contains more than one of the same type of device (e.g., DCS(2)). In a standard system, the unit number defaults to (1); thus, the (1) need not appear in the statement.

parameters the parameters allowed in an instrumentation statement depend on the particular device specified in that statement. Therefore, the list of allowable parameters is very much device-dependent and the function of each parameter in the list may differ slightly depending on the device. Table 4-3 contains a matrix showing which parameters are allowed with which instruments, followed by a summary of the basic characteristics of all instrumentation statement parameters. Unique characteristics applicable to a particular device are covered below in the description of that device.

Table 4-3
INSTRUMENTATION STATEMENT PARAMETERS

General format: command function device (unit) parameters [error-command]:

		device										
		DCS	DCM	ACZ	SCAN	MUX	SHORTS	RM	UPS	TPS	PIO	STM	I488
command	SET	X	X		X	X			X	X	X	X	X
	TEST	X	X	X			X	X	X				X
	MEAS		X	X				X	X	X	X	X	X
	CLEAR	X	X	X	X	X			X	X	X	X	X
function	DCI	X	X						X				
	DCV	X	X						X				
	RS,CS,..D,Q			X									
	R							X					
	(unit)	opt.	opt.	opt.					X	X	opt.		
parameters	AT()			X	X	X	X	X					
	V=	(DCV)		X				X	X				
	I=	X					X	X	X				
	F=			X									
	DLY=	X	X	X				X	X				
	RDLY=		X					X					
	BIAS			X									
	OFF	(DCI)											
	HIPREC			X									
	HI=,LO=,VAL=,TOL=		X	X				X					
	MAX=,FORCE		X	X				(MAX)					
	R=,VMAX=						X						
	INTO var	T	T/M	T/M			X	X	S/T/M	S/M	M	M	M
	FAIL(n)	T	T/M	T/M			X	X	X	X			
	DEVTYPE INTO								M	M			
	VA,VB,VC									S			
	OTTL(),RLY(),HRLY()										S/M		
	ITTL,SWITCH										M		
	RLY(),LED()											S/M	
	SWITCH											M	

NOTES: 1. The CLEAR statement for any device generally accepts only the device keyword and unit number (opt.).

2. In addition to being device-dependent, some parameters are also dependent on the type of statement. An X indicates that the parameter applies to all statements for that device (except CLEAR). An S,T or M indicates SET only, TEST only or MEAS only.

AT( ) Used to specify MUX or scanner connections and to specify nail lists for continuity testing. When used with the MUX and scanner, the AT( ) construct connection fields within parentheses are separated from each other by a colon (:). The first 4 fields are used for the 4 MUX output channels CHA, CHB, CHC and CHD; the fifth field, used only in MUX statements, specifies connections to the MUX reference bus. When AT( ) is used with the shorts or opens detector, the parentheses enclose a list of all nails to be tested.

Scanner connections may also be included in an ACZ or ohmmeter (RM) statement to specify UUT pins between which the impedance or resistance measurement is to be made.

Details on the format and application of the AT( ) construct are covered in the paragraphs describing the particular device using that construct (i.e., MUX, SCAN, ACZ, RM, SHORTS and OPENS).

V= Specifies the value of a source or power-supply voltage. For the ACZ, value is given in VRMS; for all other devices the value is in VDC. The ranges of allowable voltages and the default values are defined in the particular device description.

I= May either specify, in amps, the output value of a current source (or power supply) or the value of the current limit for a voltage source (or supply). The ranges of allowable currents and the default values are defined in the particular device description.

F= Frequency of the ACZ in Hz. May be 100, 1000 or 10,000 Hz. Default is 1 kHz.

DLY= Set-up delay (up to 1.999 seconds). Defines how long a source or supply is allowed to settle following a SET statement before a test or measurement is made. Can be used in a dc test to compensate for the effects of capacitance on the source node (shown as C1 in Figure 4-38) and to allow the applied voltage to reach a steady state value before proceeding.

RDLY = Specifies the delay (up to 1.999 seconds) between each range change of a dc measuring device. The use of RDLY is appropriate when a capacitance must be guarded. Current flow is affected if the capacitance on the "low" side of the resistor to be measured is appreciable. This capacitance is illustrated as C2 in Figure 4-38. RDLY forces the current measurement to be delayed before each range change until the current is stable.

BIAS= DC bias voltage for the ACZ. Allows the ACZ to operate with dc bias levels of $\pm$0.3 V or $\pm$1.5 V, in making ac measurements.

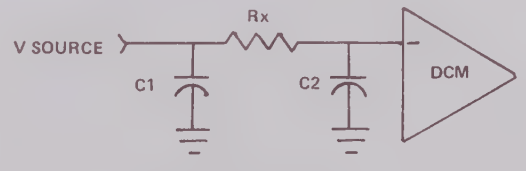


Figure 4-38. Applications for DLY and RDLY.

HIPREC — Causes the ACZ to use a longer integration period and to provide a more precise measurement. Used if more accurate results are needed.

HI=
LO=
VAL=
TOL=
(limits) — Limits may be specified using any of 4 parameters (HI=, LO=, TOL=). Single-sided limits may be defined by HI= and LO=. Double-sided limits may be specified by using both HI= and LO= or by giving the nominal value (VAL=) and tolerance limits (TOL=), where tolerance is expressed as a percentage of the value. Limits are an integral part of a TEST statement but may be used in a MEAS statement as well.

Examples:

1. LO=4.7 K means that the value should be at least 4700 ohms.
2. LO=9 K HI=11 K means that the value should be between 9 K and 11 K.
3. VAL= 10 K TOL= 10 also means that the value should be between 9 K and 11 K.

MAX=
FORCE
(ranging) — If neither ranging parameter MAX= or FORCE is specified, most system instruments will begin to measure on their next-to-least sensitive range. If the measurement cannot be made on this range, auto-ranging will take place and the best measurement range will be automatically found.

Measurement efficiency can be improved and test/measurement time reduced by properly specifying the MAX parameter. MAX is a floating-point value that defines the starting range for the measurement.

Additional control over ranging can be obtained with the FORCE parameter. FORCE disables autoranging and forces the measuring device either to the range specified by MAX or to the next-to-least sensitive range (if MAX is not used). If the measurement cannot be made on this "forced" range, a FAILure is indicated and the designated failure option is taken.

Instrument ranges are listed in the service manual for your system.

R= — Defines the resistance threshold for detecting shorts or opens between analog nails. The value of R may range from 5 to 1000 ohms, with a default value of 10 ohms. Note that R (not followed by an equal sign) may also be used to specify a function (e.g., TEST R or MEAS R).

VMAX= — Specifies the applied dc voltage during continuity testing. The value of VMAX may range up to 16 V, with a default value of 0.1 V.

INTO variable — Basically used to store a value measured by a MEAS statement. May May also be used to store a 0 (fail) or 1 (pass) indicator resulting from certain TEST statements.

FAIL Options — Any instrumentation statement that can fail may have any of several user-defined options that will cause the program to take a specific action whenever that statement fails. These options, FAIL vector, [error-command] and failure reporting, are described in Section 5.

NOPRINT — Suppresses any diagnostic reports resulting from the test.

Reacting to a Test Failure - Section 5

5.1 GENERAL

Testing implies failure. When the measured value of a component does not meet its given specification or when a test statement cannot be performed as specified, a failure occurs. An instrumentation failure can occur for any of the following reasons:

1. Limits are present and the tested value is found to be outside the limits.

2. Any parameter computed at test time is found to have an inappropriate value.

3. The requested measurement cannot be made because of unanticipated UUT behavior. This could include problems such as the current source operating in voltage compliance or current limiting of the voltage source.

4. The requested measurement cannot be made because of test system malfunction.

Digital tests can fail for similar reasons. Most commonly, a digital test fails because a component output pin fails to reach the expected logic state.

The user has several available options for specifying how the program should react to a failure.

1. The test programmer can DECLARE a fail vector to record the status of any number of failure conditions.

2. A set of failure-reporting options defining failure routines and messages can be assigned to individual statements or groups of statements.

3. An [error-command] can be included in any test statement, to be executed whenever that statement fails.

When a test statement fails, the fail vector is set first (unless explicitly inhibited by the user), then the failure reporting options are performed (message assignments and subroutine execution), and finally the [error-command] is executed (if one is specified).

5.2 FAIL VECTOR

5.2.1 General

The FAIL vector is a bit string defined by the programmer at the start of the program. At that time, all cells (bits) in the vector are reset to zero. When a test statement fails, the specified FAIL vector bit will get set to one,

unless explicitly inhibited by the programmer. At the end of the program, if any FAIL vector bit remains set, the FAIL light on the keypad is turned on. Otherwise, the PASS light is turned on.

Format

```
DECLARE BSTRING FAIL(m);
 |
TEST/MEAS ----- FAIL(n)-----;
```

where:

DECLARE BSTRING	Command keywords.
TEST/MEAS	Command keywords for intrumentation statements.
FAIL	Keyword for FAIL vector bit string.
(m)	Maximum number of cells (bits) in the FAIL vector. Must be DECLAREd at the start of the program.
(n)	The number of the FAIL cell that will be set if the instrumentation statement fails. If FAIL(n) is omitted from the statement, FAIL(1) is assumed. If FAIL( ) is specified, execution of the statement will have no effect on the FAIL vector.

Examples

```
TEST DCI DCS FAIL(7);
```

Meaning: FAIL bit 7 is set, if the current source cannot supply enough voltage to force the current for which the DCI is programmed (voltage compliance error). Because a fail bit is set, the FAIL indicator is turned on at the end of the program.

```
TEST DCI DCS [BRANCH CURRENTERR;];
```

Meaning: The default FAIL bit 1 is set if the current source is in voltage compliance. If this occurs, the FAIL indicator is turned on at the end of the program. In addition, a branch to the label CURRENTERR is taken.

```
TEST DCI DCS FAIL( );
```

Meaning: No FAIL bit is set even if the current source is in voltage compliance. A FAIL(null) can be included in an instrumentation statement as shown here to inhibit a test failure from showing a pass/fail indication. The test failure is masked, and the FAIL indicator remains off.

```
TEST DCI DCS FAIL(7);
  |
IF FAIL(7) THEN BRANCH N30;
```

Meaning: If the current source is in compliance, set FAIL bit 7. Test the status of FAIL bit 7 by means of the IF-THEN-ELSE statement. If FAIL(7) is set, indicating a failure, branch to N30.

Rules and Considerations

1. The FAIL vector should not be used in an assignment (LET) statement because this may change the FAIL declaration, resulting in erroneous operations.

2. The FAIL vector must be DECLAREd as a bit string variable at the start of the test program.

3. The parameter FAIL( ) in an instrumentation statement will prevent that statement from having any effect on the FAIL vector.

4. If FAIL(n) is omitted from an instrumentation statement, FAIL(1) is assumed.

5. Once a FAIL bit is set by an instrumentation statement, other instrumentation statements will have no effect on that FAIL bit (i.e., the FAIL bit will remain set unless explicitly cleared by BITCLR, etc).

5.3 USER FAILURE-REPORTING OPTIONS

The 227X test language provides a number of pre-defined constructs and variables as a convenience for reporting errors. These allow the user to specify various failure-reporting options that may be taken whenever an instrumentation statement fails. The system test devices supported by this feature are: DCS, DCM, ACZ, RM, ARITH, SHORTS, OPENS and BURST. These options may also be specified in a BURST (digital test) statement. The failure-reporting options may either be specified as parameters within individual instrumentation statements or they may be predefined via a USE statement. The options consist of user-defined routines and messages. The routines have access to special diagnostic variables defined by the system whenever an instrumentation statement fails.

5.3.1 Failure Options

At the start of test program execution, the values of all failure options and variables are reset (i.e., character-string variables are all set to nulls and floating-point variables are reset to zeros). These options and variables may then be assigned values for individual instrumentation statements by including the options as parameters in each statement.

For example,

```
MEAS R INTO RVAL . . . . . FSUB=FAILSUB  DMSG='NOMINAL';
```

Or they may be specified for all subsequent instrumentation statements by means of a USE statement (see USE statement description in para 6.2).

For example,

```
USE FSUB=FAILSUB  DMSG='NOMINAL';
```

The USE statement defines the default options for all subsequent FAILed statements (i.e., instrumentation statements without explicit failure parameters will revert back to the failure options specified by the most recent USE statement). When explicit failure parameters are used in a statement, the options specified by those parameters apply only to that statement.

The user failure options are as follows:

```
NOFSUB
FSUB=name-of-routine
PMSG='user-part-message'
DMSG='user-diagnostic-message'
CMSG='user-component-message'
```

where: NOFSUB

Inhibits the calling of subroutine FSUB in the event of a failure in an instrumentation statement. This is the initial default state prior to program execution.

FSUB=name-of-routine

This option can be used to replace a [CALL subroutine] error-command in an instrumentation statement. If the statement fails, the FAIL bit (if specified) is set, then the failure routine is invoked. Upon returning from the routine, the [error-command] is then executed (if one was specified). Failure routines must be DECLAREd, the same as any other subroutines, but they cannot have an argument list. Instead, they obtain information about the failure from special system failure variables, as described in para 5.3.2.

PMSG='user-part-message'
DMSG='user-diagnostic-message'
CMSG='user-component-message'

Each of these 3 user-defined messages are made available to the failure routines. Each message can be 80 characters long and, like the failure routines, may be specified in a USE statement or as parameters in an instrumentation statement.

5.3.2 Failure Variables

The system variables listed in Table 5-1 provide access to test result information in the event of a detected UUT or system instrument failure.

When an instrumentation statement fails, new values are assigned to the failure variables, as appropriate. Failure variables not applicable to the statement are reset to nulls or to zeros.

Other rules and considerations associated with using these variables include:

1. Do not DECLARE system failure variables.

2. These variables cannot be read into by an I/O instruction or by the INTO parameter of a TEST/MEAS statement, and they cannot be assigned a value via a LET statement.

3. These variables can, however, be tested (e.g., as part of the IF condition in an IF-THEN-ELSE statement).

Table 5-1
CHARACTER-STRING VARIABLES

FTYPE - Two-character string indicating what quantity was tested/measured in the last FAILed statement.
- R - Resistance
- V - Voltage
- I - Current
- RP - Parallel Resistance
- RS - Series Resistance
- CP - Parallel Capacitance
- CS - Series Capacitance
- LP - Parallel Inductance
- LS - Series Inductance
- XP - Parallel Reactance
- XS - Series Reactance
- ZM - Impedance Magnitude
- QQ - Quality = XS/RS = RP/XP
- DD - Dissipation = RS/XS = XP/RP
- VS - Voltage Source (TEST DCS DCV)
- IS - Current Source (TEST DCS DCI)
- AR - Arith (TEST ARITH)
- DG - Digital

FSTAT - Indicates the status of the test results for the last FAILed instrumentation statement:
- *LOW - Measured value was low,
- *HIGH - Measured value was high,
- *FAIL - Measurement failure
- #ERR - Testing error occurred.

FLABEL- Label associated with the last FAILed instrumentation statement, i.e., the last label or subroutine used before the failure.

FDMSG - DMSG associated with the last FAILed instrumentation statement.

FPMSG - PMSG associated with the last FAILed instrumentation statement.

FCMSG - CMSG associated with the last FAILed instrumentation statement.

FMEAS - Measured value from the last FAILed instrumentation statement.

FHI - HI limit value (if available) from the last FAILed instrumentation statement, or some value greater than 1.0E30 (to indicate that no HI limit is available).

FLO - LO limit value (if available) from the last FAILed instrumentation statement or some value less than -1.0E30 (to indicate that no LO limit is available).

NOTE

The failure variables reflect the values associated with the most recent FAILed instrumentation statement.

5.3.3 System Message File

Failure messages and current values of the failure variables can be written in the system message file via the following statement:

```
WRITE ID=MESFILE data ;
```

where: WRITE — Command keyword.

ID=MESFILE — Specifies the I/O channel for the system message file which is defined by the MESsage and ID=MES file device (.MES) option on the DIAGNOSE monitor page. This MESFILE channel should not be explicitly OPENed or CLOSEd.

data — The same format as any other WRITE statement (refer to WRITE description in para 6.2).

5.3.4 Examples of Failure-Reporting Routines

1.
```
DECLARE FAILSUB( );
     '
     '
SET MUX AT (6:6:8:8:5,6,8,11);
USE FSUB=FAILSUB;

/*TEST PIN A89*/
SET SCAN AT (117:117:89:89);
MEAS R INTO RVAL V=.1 HI=10 MAX=1 CMSG='PIN A89'
     DMSG='OPEN FROM 117 TO 89';

/*TEST PIN A57*/
SET SCAN AT (117:117:57:57);
MEAS R INTO RVAL V=.1 HI=10 MAX=1 CMSG='PIN A57'
     DMSG='OPEN FROM 117 TO 57';

     '
     '
```

Meaning: Connect the MUX for a group of tests and define FAILSUB as the user failure routine. Measure resistance between pin 89 and 117. If the resistance is greater than 10 ohms, the MEAS statement will fail, causing a transfer to FAILSUB and assigning the indicated character string values to CMSG and DMSG.

2.
```
     '
(from Example 1)
     '
/*START RESISTANCE TEST*/
SET MUX AT (6:8:9:10:5,6,11);
USE DMSG='NOMINAL';

/*TEST R6*/
SET SCAN AT (135:134);
MEAS R INTO RVAL V=200M HI=4.9K LO=4.4K CMSG='R6';
     '
     '
```

Meaning: Connect the MUX for a 2-terminal resistance test. Define a new default diagnostic message DMSG='NOMINAL', then measure R6. If the measured resistance falls outside of limits LO=4.4K and HI=4.9K, the statement will fail. User failure routine FAILSUB will be called; CMSG='R6' and DMSG='NOMINAL' will be defined.

3. SUBROUTINE FAILSUB( );
```
WRITE ID=MESFILE '%S% FAILED %S%%S%' FCMSG,FTYPE,FSTAT;
WRITE ID=MESFILE '%S% MEASURED:%5G%' FDMSG,FMEAS;
```

Meaning: Store the indicated data in the system message file. This message file is specified by the MESsage device (.MES) option on the DIAGNOSE monitor page. The definitions of failure variables FMSG, FTYPE, etc are given in para 5.3.2. The values assigned to these variables are those associated with the most recent FAILed statement.

5.4 ERROR-COMMANDS

The [error-command] construct can be used in an instrumentation statement to specify one or more commands that will be executed if the statement cannot be performed or if a test fails. This error-command may be either a simple statement (single command) or a compound statement (multiple commands).

Example

```
TEST Z LO=550 FAIL(5) [CALL ZFAIL();];
```

Meaning: If the measured impedance (magnitude) is less than 550 ohms, set FAIL bit 5, then execute subroutine ZFAIL.

5.5 BURST ERRORS

Like analog measurements, digital tests can also fail. Whenever a sensor detects a logic state different from the one that was expected (specified by OH or OL), the test step fails. In addition, the burst as a whole is considered to have failed. Because the digital testing software includes automatic error reporting, special error-handling subroutines are usually not required for digital test bursts. Nevertheless, if necessary, the END BURST statement can include a jump to an error-handling subroutine, with whatever parameter-passing statements are required. The same pre-declared fail variables used in analog testing (Table 5-1) are available, but most are meaningless in reference to a burst and should not be used.

Test Language Descriptions – Section 6

6.1 GENERAL

This section describes all of the functions, operators and other keywords used for UUT testing. These are subdivided into 3 groups: (1) General Utility statements, (2) Analog Instrumentation statements, and (3) Digital Test statements. Table 6-1 lists all of the statement keywords in the test language (without parameters).

Table 6-1
TEST LANGUAGE KEYWORDS

Utility Statements & Functions	Analog Instruments	Digital Testing: All Systems	Digital Testing: SpeedPlus Only
BITCLR	ACZ	ASSIGN LGC	CLEAR CLOCK
BITSET	ARITH	BREAK	CLKGND
BRANCH	DCM	BURST/END BURST	DECLARE NAILBIND
CALL	DCS	BUSTEST/END BUSTEST	FAST/END FAST
CLEAR	I488	DECLARE	FEXCLOSE
CLOSE	MUX	CLEAR LGC	FEXGND
CONCATENATE	OPENS	DECLARE GROUP	FEXOPEN
DATE( )	PIO	EXITIF	FLAGFAIL
DECLARE	RM	GOSUB (not in FAST)	FLOOP/END FLOOP
DELAY	SCAN	GOTO	FNAILOFFSET
END	SHORTS	LOOP/END LOOP	FSCNCLOSE
EXIT	STM	LOOPIF	FSCNOPEN
INDICT	TPS	OR	GOTO FLAG FAILS
IF-THEN-ELSE	UPS	RETURN	GOTO FLAG PASSES
KEYBD( )	HCS	SET LGC	GOTO NOPHASELOCK
LENGTH( )		SUBROUTINE	GOTO PHASELOCK
LET		/END SUBROUTINE	RLYCD
LOAD/UNLOAD		TEST LGC	RLYCDEX
MEAS			RLYCS
NAILBIND			RLYSEX
NAILOFFSET			RLYT
ON ABORT			RLYTEST
OPEN			RLYTEX
PAUSE			SET CLOCK
PEXCLOSE			TEST CLOCK
PEXGND			TRGGND
PEXOPEN			
READ			
RETURN			
SCNCLOSE			
SCNOPEN			
SDSOPT			
SET			
SUBROUTINE/END SUBROUTINE			
SUBSTR( )	USE		
TEST	WHILE-DO		
TIME( )	WRITE		

Test language functions are described in a format consisting of the following major items:

General - Describes the purpose of the statement and how it is used to control and monitor the test system.

Format - Defines the specific construct of the statement and all allowable parameters. System keywords are shown in UPPER-CASE, and user-assigned inputs are shown in lower-case.

Example - Illustrates typical usage of that statement in a test program.

Rules and Considerations - Lists various guidelines and constraints to be considered when using the statement in a test program.

6.2 GENERAL UTILITY STATEMENTS

This section covers all general-purpose utility statements such as input/output, flow control, data manipulation, etc. Note that these statements cannot be used within a digital burst -- only outside a burst. Some general keywords are used also as burst keywords, but they have different meanings within a burst.

----------BITCLR----------

General

This command is used to reset any one bit in a bit string. It is identical to the BITSET command, except that it resets rather than sets the specified bit.

Refer to the BITSET description for examples, rules and considerations applicable to BITCLR.

Format

```
BITCLR (bit-string-name, bit-number);
```

where:	BITCLR	Command keyword.
	bit-string-name	Name of the bit-string variable being altered.
	bit-number	Position of the bit to be cleared. Position 1 refers to the left-most bit in the string. May be a floating point number or variable.

Rules and Considerations

1. BITCLR is used to reset single bits in a bit string. To reset all bits in the string at once, use "LET bit-string-name=0;".

----------BITSET----------

General

This command is used to set any one bit in a bit string. It can operate on any previously DECLAREd bit string variable, such as the FAIL vector (refer to para 5.2) or a scanner variable (see SCAN description in para 6.3). BITSET, in conjunction with BITCLR, is especially convenient for changing scanner pins without altering the AT( ) construct in an instrumentation statement. This feature is illustrated in the following examples:

Note that BITSET sets only the specified bit in the bit string and leaves all other bits unchanged.

Format

```
BITSET (bit-string-name, bit-number);
```

where:

BITSET	Command keyword.
bit-string-name	Name of the bit-string variable being altered.
bit-number	Position of the bit to be set. Position 1 specifies the left-most bit. May be a floating-point number or variable.

Examples

1.
```
DECLARE BSTRING BSA(256);
DECLARE BSTRING BSB(256);
        '
        '
BITSET (BSA, 15);
BITSET (BSB, 16);
        '
        '
SET SCAN AT (BSA:BSB);
```
(Note: This example applies to a 2270 system only.)

Meaning: DECLARE 2 bit strings (BSA and BSB), then set bit 15 in BSA and bit 16 in BSB. Leave all other bits unchanged. The SET SCAN statement will connect channel A to scanner pin 15 and channel B to pin 16.

2.
```
DECLARE BSTRING BSC (256);
DECLARE I;
    '
    '
LET I=6;
WHILE I<256 DO
[BITSET (BSC,I);
SET SCAN AT (5:BSC);
    '
    '
BITCLR (BSC,I);
LET I=I+1;];
```

Meaning: Use the variable I to specify bit position in the BITSET statement. Connect scanner pin 5 to channel A and the scanner pin defined by BSC to channel B. (Initially, channel B gets connected to scanner pin 6 since the initial value of I is 6.) Once the desired testing with this scanner pin configuration is completed, reset the BSC bit via a BITCLR command, increment I, and set the next BSC bit. This now connects scanner pin 7 to channel B.

Connect successive scanner pins to channel B until I=256, then terminate the loop.

Rules and Considerations

1. Each BITSET statement can set only one bit in the specified bit string.

2. All other bits remain unchanged.

----------BRANCH----------

General

The BRANCH statement causes an unconditional transfer to a specified statement in the program.

Format

```
BRANCH label;
```

where: BRANCH Command keyword.

label Identifies the test program statement to which the branch is made.

Examples

1. BRANCH ERR1;

Meaning: Transfer program control to the statement labeled ERR1.

2.
```
          TEST SHORTS AT (2,4,6) [BRANCH FSHORT;];
          WRITE 'NO SHORTS DETECTED';
          BRANCH ESHORT;
FSHORT:   WRITE 'FAILED SHORTS TEST';
ESHORT:   END;
```

Meaning: If shorts test is successful, print out NO SHORTS DETECTED, then branch to the END statement. If shorts test failed, branch to FSHORT and print out FAILED SHORTS TEST.

----------CALL----------

General

The CALL statement causes a program transfer to the indicated subroutine. Once subroutine execution is complete, program control automatically returns to the next instruction following the CALL statement. CALL can only be used for calling subroutines outside a burst. External routines (described in Appendix C of this manual) are CALLed in exactly the same way as internal routines.

Format

```
CALL name (argument-list);
```

where:

CALL	Command keyword.
name	Subroutine name as it appears in the SUBROUTINE definition and declaration.
argument-list	Assigns values to the list of local variables used in the subroutine. The argument list takes the form: (parameter1=argument1,.....)

Example

```
1.  DECLARE FAILR (FLOAT MVAL, FLOAT VALUE);
                   '
                   '
    CALL FAILR (MVAL=RESULT, VALUE=1K);
```

Meaning: Jump to subroutine FAILR and define the subroutine parameters as indicated. Test program value RESULT is assigned to subroutine parameter MVAL, and VALUE is defined as 1K.

Rules and Considerations

1. Arguments in a CALL statement are referenced by keyword rather than by position, and they can therefore be listed in a different order than in the DECLARE statement.

2. The CALL statement must include the parentheses, even if no arguments are being transferred to the subroutine (e.g., CALL FAILR( )).

3. A subroutine is not called by value, but by reference (i.e., a pointer to the argument is passed into the subroutine). As a result, it is possible for subroutines to modify variables in the calling program.

4. An external subroutine is called in exactly the same way as an internal subroutine; the preceding declaration must, however, include the keyword EXTERNAL (refer to DECLARE). External routines can be prepared only on systems that can compile and link programs written in a language other than 227X test language. Refer to Appendix C.

----------CLEAR device----------

Refer to description of the specific device being addressed by this statement (i.e., ACZ, DCM, DCS, etc).

----------CLOSE----------

General

Each channel explicitly opened by an OPEN statement must be closed by a CLOSE statement prior to the end of the program.

Format

CLOSE ID=channel;

where: CLOSE Command keyword.

ID=channel ID is the keyword; channel is the user-assigned name given to the channel in a previous OPEN statement.

Example

1. CLOSE ID=APPLE;

Meaning: Close the channel identified as APPLE by a previous OPEN statement.

----------CONCATENATE (OPERATOR) !!----------

General

The !! operator allows 2 character strings to be concatenated (i.e., one string appended to another) into a new string.

Format

string 1 !! string 2

where: string1, string2 The name of the character strings to be concatenated.

!! Operator symbol, used to concatenate strings, i.e., to append one string to another.

Example

```
1.  DECLARE CSTRING Y1(10);
    DECLARE CSTRING X1(5);
    DECLARE CSTRING X2(5);
    LET X1='ABC';
    LET X2='DEF';
     '
     '
     '
    LET Y1=X1 !! X2;
```

Meaning: The new character string Y1 will be ABCDEF.

Rules and Considerations

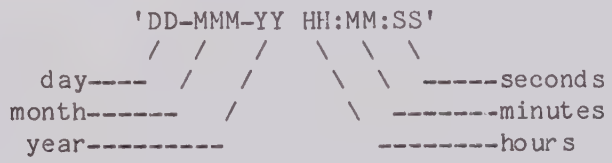
1. The concatenate operator (!!) can only appear as part of a LET statement.

2. The DECLAREd length of the new string must be large enough to accommodate the combined length of the 2 substrings; otherwise, characters will be lost.

---------DATE( )---------

General

The DATE( ) function is used to retrieve date and time from the operating system. This information can then be printed out as part of user-generated messages. The format of the returned character string variable is as follows.



Format

```
DATE( )
```

Example

```
1.  LET NOW=DATE( );
    WRITE '%S%' NOW;
```

Meaning: Store the date in character string variable NOW, then display it on the screen. For example,

```
07-JUL-80 14:43:35
```

Rules and Considerations

1. DATE( ) is a function and thus cannot be used alone but only as part of a statement.

2. The DATE( ) function cannot be used directly as a parameter in a WRITE statement (e.g., WRITE '%S%' DATE( );). The DATE ( ) function should first be assigned to a variable as shown in Example 1.

----------DECLARE----------

General

Declaration statements are necessary to define all user-named variables and subroutines used in the test program. As described below, several types of variables and subroutines may be DECLAREd:

A. Variables:

1. Local variables. These are known only within the program block (main body or subroutine) in which they are DECLAREd.

2. Global variables. These are known throughout the test program. Global variables permit values to be passed to a subroutine without explicitly stating the variables in the argument list of the subroutine. The global variable is also particularly useful in a user-failure routine invoked by FSUB, because there is no provision for arguments in a user-failure routine. The FAIL vector is considered a global variable, although it cannot be DECLAREd as such.

3. Load variables. These have the same scope as global variables, but remain defined from program load to program unload time. Local and global variables, on the other hand, are redefined at each program execution.

4. Digital pin groups. For convenience in writing digital test programs, variables can also be used to define groups of digital test pins. (Refer to DECLARE GROUP statement).

B. Subroutines:

1. Internal Subroutines. These are the normal subroutines written in 227X test language and included in the .TPG file following the main body of the test program.

2. External Subroutine. These routines are written in a language other than 227X test language (e.g., MACRO-11) and are not part of the test program file. External routines are independent programs installed as separate tasks on the 227X system. Once installed in the system, however, they may be CALLed in the same way as internal routines. (See Appendix C.)

3. Load/Unload Subroutines. Internal subroutines DECLAREd as load (and unload) routines are automatically invoked by the system at program load (and unload) time.

Format

1. Test Program Variables

   a. Local: DECLARE type name(length);

   b. Global: DECLARE GLOBAL type name(length);

2. Subroutines

   DECLARE subroutine-name(argument-list);

3. External Subroutines

   DECLARE EXTERNAL subroutine-name(argument-list);

where: DECLARE/GLOBAL/EXTERNAL — Command keywords.

type — Keyword defining the variable type. May be any of the following:

Type	Example	Keyword
Bit string	1001101	BSTRING
Character string	R1 FAILED	CSTRING
File	LB1:TEST2.TPG	FILE
Floating point	34.17	FLOAT
Digital pin group	see DECLARE GROUP	GROUP

If no type specification is made, the variable is assumed to be floating-point.

name — User-assigned variable name.

(length) — A decimal number specifying the maximum length (number of bits or characters) in a bit string or character string.

subroutine-name — User-assigned name of the subroutine being DECLAREd. If the subroutine is EXTERNAL, this is the name under which the subroutine has been installed as a task on the system.

(argument-list) — Lists and defines the type of each variable passed to the subroutine. If no variables are passed, the left and right parentheses must still be included.

Examples

1. DECLARE X;

   Meaning: X is defined as a local floating-point variable.

2. DECLARE GLOBAL Z;

   MEANING: Z is defined as a global floating-point variable.

3. DECLARE FILE LOG;

   Meaning: LOG is defined as a file variable to be used as the channel identifier in an OPEN statement.

4. DECLARE BSTRING FAIL(18);

   Meaning: The FAIL vector is defined as a bit string, up to 18 bits long. Note that the FAIL vector must always be DECLAREd and must have a minimum specified length of 1. (See FAIL vector description in para 6.5).

5. DECLARE FAILR (FLOAT MVAL, FLOAT VALUE);

   Meaning: The argument list for subroutine FAILR defines MVAL and VALUE as floating-point variables.

6. DECLARE EXTERNAL RSHIFT(BSTRING LATCHIN(16), CSTRING PLACE(1));

   Meaning: The program RSHIFT, written with an interface to the 227X external subroutine call, may be called from this program. The parameters LATCHIN, with a length of 16 bits, and PLACE, with a length of one character, may be passed to or from the external subroutine.

Rules and Considerations

1. All variables (except system variables) and subroutines must be DECLAREd.

2. If a string exceeds its DECLAREd length, all overflow bits or characters will be lost.

3. DECLARE statements for both local and global variables must appear before any other executable statement in the test block in which those variables are used.

4. A global variable and a subroutine can be DECLAREd any number of times. It is recommended that global variables and subroutines be DECLAREd in every program block (main body or subroutine) that references those variables.

5. If a local and global variable both have the same name within a program block, the local variable takes precedence.

6. A DECLARE subroutine( ) must have the parentheses, even when no arguments are passed to the subroutine.

7. Although the FAIL vector is a global variable, it is DECLAREd without the keyword GLOBAL, and it can only be DECLAREd once during the program.

----------DECLARE GROUP----------

General

Allows the user to define groups of digital pins that can later be set to a desired state via a numeric quantity. Taking a 12-bit memory address as an example, it is easier and more natural to specify the address as a 3-digit hexadecimal number (or a 4-digit octal number) rather than to specify the state of all 12 bits. DECLARE GROUP assigns a group name to the specified pins and allows that collection of pins to be referenced by a single group name. Refer to para 6.4.3 - Logic Pin Specifications for a description of pin specifications (IC, OI, IG, etc).

Format

DECLARE GROUP group-name (pin-list);

where: DECLARE GROUP Command keywords. Must appear first in the statement.

group-name Name assigned by the user to the group of pins.

pin-list An ordered list of numbers representing the pins assigned to that group. The first (leftmost) entry corresponds to the most significant bit and the rightmost bit corresponds to the least significant bit.

Once DECLAREd, the group-name may be used in either of the following formats:

1. Ix or Oy or Pz (group-specs, pin-specs);

where: Ix Allowable specs for input pins (i.e., IC, ID, II, IS, IH or IL).

Oy Allowable specs for output pins (i.e., OI, OS, OH, OL, OM, or OU).

Pz Allowable specs for terminating resistors (i.e., PU, PD, PN or PB).

group-specs One or more group-names previously defined in a DECLARE GROUP statement. Names must be separated by a comma.

pin-specs Optional. Individual pin assignments may also be included, each separated by a comma, e.g., OS(REG1, REG2, 4, 7, 9).

2. IG or OG (group-values);

where: IG DECLAREd group of input pins.

OG DECLAREd group of output pins.

group-values Used to assign values to previously defined group-names. For example: OG(OUT1=17, OUT2=23); In addition to numerics, group names may also be used to specify values, e.g., OUT3=XYZ.

Examples

1. DECLARE GROUP OUT(11-14);

   Meaning: Assign the variable name OUT to pins 11-14. If a value is later defined for this pin group, pin 11 represents the most significant bit of that value and pin 14 represents the least significant bit.

2. DECLARE GROUP OUT(14-11);

   Meaning: Similar to example 1, but pin 14 represents the most significant bit and pin 11 the least significant bit.

3. With pin group OUT DECLAREd as shown in the previous example, pins 11, 12, 13 and 14 can all be set high by any one of the following statements:

```
OH(11-14);
OH(OUT);
OG(OUT=B'1111');
OG(OUT=O'17');
OG(OUT=D'15');
OG(OUT=X'F');
```

4. DECLARE GROUP REG(8,2,14,17-15);

   Meaning: Assign the variable name REG to the series of pins, where pin 8 represents the most significant bit, and pins 2, 14, 17, 16 and 15 represent the less significant bits in that order.

5. OG(OUT=OUT+1);

   Meaning: Increment the present value of OUT by 1.

6. OG(OUT=IN);

   Meaning: The present value of IN is assigned to the group of pins defined by OUT. If necessary, the value is right-justified, with leading-zeros inserted if IN specifies fewer pins than OUT, or with the most significant bits truncated if IN specifies more pins than OUT.

Rules and Considerations

1. Group names may be up to 11 characters long.

2. Each group may have a maximum of 15 pins.

3. Values within IG and OG lists can be specified using compile-time arithmetic, as shown in Example 5. Note that only group names and constants are allowed in an expression.

4. DECLARE GROUP must be outside the burst, but group type specifications may only be used within a burst.

5. The statement of Example 5 would not repeatedly increment the value of group OUT if the statement were in a LOOP. The context of the group OUT will always be the original value when the LOOP is re-entered.

----------DECLARE NAILBIND---------

General

The NAILBIND declaration is used to allow digital or analog testing at any of the timing nails belonging to the optional SpeedPlus subsystem. These clocking and trigger nails do not, by themselves, allow normal testing. Directly connecting a driver/sensor nail to a timing nail would degrade the timing nail's performance. To avoid permanent extra loading, each timing nail has an extra relay that can connect it to a test nail when necessary. If digital or analog tests must be conducted at the timing nail, the user installs a jumper in the test fixture from a free test (driver/sensor) nail to this relay, and informs the system of the connection with a DECLARE NAILBIND statement. The system then closes the relay when digital or analog tests are specified at the timing nail. Tests specified at the bound test nails are then actually performed at the specified timing nails.

Format

DECLARE NAILBIND (nail-list=timing-nail) (nail-list=timing-nail) . . .;

where: DECLARE NAILBIND Command keywords. Appear only once, at the beginning of a test program.

nail-list A list of test nails separated by commas. All test nails named in the nail-list will be treated by all analog and digital test statements as equivalent to the named timing-nail. Both standard and SpeedPlus test nails can be named.

timing-nail A single clock driver (CD), clock sync (CS), or trigger nail (T) in the SpeedPlus subsystem at which digital or analog tests will be conducted.

Example

1. DECLARE NAILBIND (1=CD1) (2=CD2) (4001=T1) (4002,4378=T2);

   Meaning: Test nails 1, 2 are wired to the external connection relays of clock drive nail 1 and 2 (CD1, CD2); (SpeedPlus) test nail 4001 is wired to trigger nail 1 (T1); and test nails 4002 and 4378 are wired to trigger nail 2 (T2). Tests specifying test nails 1, 2, 4001, 4002, or 4378 will be carried out at the bound timing nails.

Rules and Considerations

1. DECLARE NAILBIND must precede all executable statements in the program.

----------DELAY----------

General

The DELAY statement suspends test program execution for the specified time interval. Once that delay interval is over, program execution is automatically resumed.

Format

```
DELAY seconds;
```

where: DELAY — Command keyword.

seconds — The delay interval specified in seconds. Standard scale factors U(Micro), M(Milli), etc can be used.

Example

```
1. WRITE ID=I488(1) '.........' [BRANCH ERR;];
   DELAY .05;
   TEST I488 [BRANCH ERR;];
```

Meaning: Delay 50 ms after sending a message to the I488 bus, before testing the Service Request (SRQ) line. Branch to ERR if device does not respond properly.

Rules and Considerations

1. There is no practical specification limit on the value of the delay interval.
2. The DELAY statement should not be confused with the special-purpose DLY (and RDLY) delay parameters used with the DCS and DCM.
3. DELAY produces a 5-ms minimum delay due to overhead.

----------END----------

General

END indicates to the translator that the end of the test program has been reached. An END statement must terminate each test program. It resets the hardware and controls the PASS and FAIL indicators. If any FAIL bit is set, END turns on the red FAIL light. If the FAIL bit string is all zeroes, the PASS light is turned on.

Format

```
END;
```

Rules and Considerations

1. END should not be confused with EXIT, although both control the PASS and FAIL indicators during testing. During compiling time, END is required to inform the translator that the last statement in the test program has been reached.

2. As with other statements, an END statement may have one or more labels (e.g., L1:L2: END;).

----------EXIT----------

General

The EXIT statement is used in a test program to stop further device testing. The test hardware is cleared (set to an initialized state). Additionally, if the FAIL bit string contains any non-zero bits, the red FAIL lamp is turned on at the end of the program. Conversely, if there are no FAIL bits, the green PASS lamp is turned on.

Format

```
EXIT;
```

Examples

1.
```
TEST SHORTS AT (5,6,10,12)
[
WRITE 'FAILED SHORTS TEST';
EXIT;
];
NOSHORT:
```

Meaning: If the shorts test failed, display the message FAILED SHORTS TEST, then EXIT from the test program. If the shorts test is successful, execute the statement labeled NOSHORT.

----------FNAILOFFSET----------

Refer to NAILOFFSET description.

----------IF-THEN-ELSE----------

General

The IF-THEN-ELSE statement is a conditional statement used for decision making. It specifies under what conditions specified statements or statement sequences will be performed. If the designated condition is met, statements following THEN will be executed. If the condition is not met, statements following ELSE will be executed.

Format

```
IF condition THEN statement; ELSE statement;
```

or
```
IF condition THEN
[
statements;
];
ELSE
[
statements;
];
```

where: IF-THEN-ELSE Command keywords.

condition — Specifies under what condition the statement will be performed. The statement is executed if the condition is true. May be either an expression (e.g., A/B) or a Boolean variable that can assume a true or false value.

statement — Any test language statement (or statement sequences except burst-only types). The ELSE statements are optional. If multiple statements are used, a left bracket ([) must precede the first statement and a right bracket (]) must follow the last statement terminated by (;). All statements enclosed within the brackets must be terminated by a semicolon (;), including the last statement.

Examples

1.
```
IF V11 - V12 > 0.5 THEN WRITE 'DIODE INSERTED WRONG';
```

   Meaning: If the difference between variables V11 and V12 is greater than 0.5, then the WRITE error message is displayed on the terminal. If the difference is less than 0.5, then the next statement is executed.

2.
```
IF V12 < V11 THEN MEAS DCI INTO I5;
```

   Meaning: If the current value for variable V12 is less than that for V11, then a current measurement is made with the results stored in variable I5. If V12 is equal to or greater then V11, then program control is directed to the statement following the IF.

3.
```
IF A > B THEN LET Z=A; ELSE LET Z=B;
```

   Meaning: After this statement is executed, the variable Z contains the maximum of either A or B.

4.
```
IF FAIL(7) THEN
[WRITE 'DIODE TEST FAILED';
BRANCH N30;];
```

   Meaning: If fail bit 7 is set, write the indicated message, then branch to N30. If bit is not set, execute the next sequential instruction after the closing bracket.

Rules and Considerations

1. Either a single statement or multiple statements may be used as the object of an IF-THEN-ELSE instruction.

2. A multiple statement sequence must be enclosed within a left and a right bracket ([ ]) and terminated by a semicolon (;).

3. Each statement in a multiple statement sequence (including the last statement) must terminate with a semicolon (;).

4. [ ] is a null statement.

----------INDICT----------

General

An INDICT statement allows the programmer to add a digital IC to the probe data base. An IC included in an INDICT statement is treated the same as an IC that failed a burst test. If probing is enabled, the user is directed to probe both failed and INDICTed ICs, with the results appearing in the digital diagnostic report. This allows users to customize their own probing routines and provides them with a greater flexibility in diagnosing UUT failures.

Format

```
INDICT 'ic-name';
```

where: INDICT Command keyword.

'ic-name' Name of the IC to be probed, enclosed by tick (') marks.

Example

1. INDICT 'U31';

   Meaning: After all failed ICs have been probed, direct the user to probe U31. A system-supplied error message will be printed that includes connection information from the .IDD file for IC31 if the .IDD file is available.

Rules and Considerations

1. As applicable to all probing, the specified IC must be in the .IDD file, that is, it must be in the ATG circuit description.

----------KEYBD( )----------

General

The KEYBD( ) function returns to the program either 0, 1 or 2 characters, depending on which key is pressed. If no key is struck, no character is returned. If a standard (non-special) character key is pressed (e.g., A, B, 1, 2, etc), the single character associated with that key is returned. If a special key is pressed, a 2-character string is returned as shown in the following table. If more than one key is pressed, only the character(s) associated with the first key struck will be returned. As soon as the returned character is read or tested, the return value is reset to null.

A typical application of the KEYBD function is to provide the user with some control over program flow by letting the keyboard keys be used as sense switches.

All standard characters are transformed to upper-case and all special keys are transformed as shown in the table. The character string is returned without the user having to press the RETURN key, as would be the case in a READ statement.

Keypad	Keyboard	Character String
YES		'Y'
NO		'N'
LOAD PROGRAM		'LO'
CLEAR ENTRY		'%177%'
RESET		See Note
MONITOR ENABLE		See Note
ENTER		'%012%'
START/CONT		'%015%'
null		''
	<left-arrow>	'LE'
	<right-arrow>	'RI'
	<up-arrow>	'UP'
	<down-arrow>	'DO'
	DEL CHAR	'%177%'
	HELP	'HE'
	DEL LINE	'%371%'
	RESTORE	'RE'
	NEXT	'NE'
	MARK	'MA'
	SRCH	'SE'
	SAVE	'SA'
	RESET	See Note
	CTRL C	See Note
	CTRL M<lf>	'%012%'
	CTRL J<cr>	'%015%'
	CTRL H	'%177%'
	CTRL U	'%371%'
	null	' '

NOTE The MONITOR ENABLE, RESET and CTRL C keys cannot be used for the KEYBD( ) function, since these keys will halt the test program.

Format

```
KEYBD( )
```

Examples

```
1.  R8: MEAS R INTO RVAL;
        WRITE 'POT R8 = %G% OHMS' RVAL;
        DELAY 2;
        IF KEYBD( ) = '' THEN BRANCH R8;
```

Meaning: Measure potentiometer R8 and display the measured value. Wait 2 seconds. If no keyboard or keypad key has been pressed, repeat the

process. When the user strikes any key (except ESC, CTRL, TAB, SHIFT), fall out of the loop and continue with the rest of the program.

2. R8: LET TEMP=KEYBD( )
   IF TEMP='' THEN BRANCH R8;
   IF TEMP='Y' THEN ...;
   IF TEMP='N' THEN ...;

   Meaning: The program loads the character string into TEMP and loops if it is a null value (indicating no key pressed yet). When any other value is returned, the program tests for a "Y" and then for an "N".

Rules and Considerations

1. The user should not use the MONITOR ENABLE, RESET or CTRL C keys for the KEYBD( ) function, since these keys halt the program.

2. As soon as the returned character string is read, it is reset to a null value. Thus, to make several tests on a single returned character string, it should be loaded into another variable first; then the variable can be tested as often as necessary.

----------LENGTH (function)----------

General

The LENGTH function determines the actual number of characters currently stored in a specified character string. This function is particularly useful in examining character strings from an IEEE-488 bus device, where the length of the string may not be known.

Format

LENGTH (char-string-name);

where: LENGTH — Function keyword.

char-string-name — The name of a DECLAREd character string.

Example

1. LET Z=LENGTH(Y1);

   Meaning: If Y1 has been previously defined as character string ABCDEF, then the value of Z will be 6, indicating that Y1 currently contains 6 characters.

Rules and Considerations

1. Since LENGTH is a function, it cannot be used alone but only as part of a test language statement.

----------LET----------

General

The LET statement allows a variable to be assigned the value of another variable, an expression or a constant. Common applications include initializing variables, performing calculations, incrementing or decrementing a count, etc.

Format

```
LET x = y;
```

where: LET Command keyword.

x A variable assigned the value y.

y A constant, variable or expression that will be evaluated if necessary and assigned to x.

Examples

1. `LET GAIN = E0/E1;`

   Meaning: The expression E0/E1 is evaluated and the resulting value is assigned to the variable GAIN.

2. `LET RNUM = RNUM+1;`

   Meaning: Increment the variable RNUM by 1.

3. `LET CS1 = 'RESISTOR TEST';`

   Meaning: Character string RESISTOR TEST is stored in CSTRING variable CS1.

4. `LET BS1 = 1,3,5;`

   Meaning: Set bits 1, 3 and 5 in bit string variable BS1 to 1.

5. `If R1 + R2 > 1050 THEN LET BIG = 7;`

   Meaning: Used as part of a flow-control statement. If the sum of variables R1 and R2 is greater than 1050, then assign the value 7 to variable BIG.

--------LOAD/UNLOAD--------

General

LOAD variables are initialized when the program .OBC file is loaded and continue to exist during all subsequent program executions, without being reinitialized or modified by the system. Since their life span is from program LOAD time to program UNLOAD time, they can be used to communicate information between program executions. Typically, LOAD variables may be used to keep track of the number of boards tested for each revision number of each board that has been revised more than once.

Users can also define a LOAD (and UNLOAD) subroutine, which the system automatically invokes at program LOAD (and UNLOAD) time. Typical uses of these subroutines are to setup the LOAD variables or to provide special instructions to the operator (e.g., clean up the test area after testing is complete).

Format

```
    DECLARE LOAD type variable-name(length);
or
    DECLARE LOAD subroutine-name( );
or
    DECLARE UNLOAD subroutine-name( );
```

where: DECLARE,LOAD,UNLOAD Command keywords.

type — Keyword defining the variable type. May be:

BSTRING - bit string
CSTRING - character string
FLOAT - floating point

If no type is specified, floating-point is assumed. (Refer to DECLARE statement description for examples.)

variable-name — User-assigned name of the LOAD variable.

(length) — A decimal number specifying maximum number of bits or characters in a bit string or character string.

subroutine-name — User-assigned name of the LOAD (UNLOAD) subroutine.

Example

```
1.  DECLARE LOAD REVLEV;
    DECLARE LOAD LSUB( );
        '
        '
    END;

    SUBROUTINE LSUB( );
    WRITE 'SPECIFY BOARD REVISION LEVEL';
    READ '%S%' REVLEV;
    END SUBROUTINE LSUB;
```

Meaning: During a program load, subroutine LSUB is called, causing the message SPECIFY BOARD REVISION LEVEL to be displayed on the screen. When the operator enters the revision number and presses RETURN, that number is stored in variable REVLEV, where it remains until the program is unloaded.

Rules and Considerations

1. The system initializes LOAD variables only during a program load operation.

2. Presently, load variables have the same scope as global variables (i.e., they are known throughout the program).

3. LOAD and UNLOAD subroutines cannot specify argument lists (but do require the parentheses associated with an argument list).

4. Only one subroutine can be DECLAREd as a LOAD subroutine, and one as an UNLOAD subroutine.

5. The system automatically calls the LOAD (and UNLOAD) subroutine during a program LOAD (and UNLOAD) operation. Thus, no explicit CALL to those routines is needed at the start or end of a test program. If the need arises, however, these 2 routines can be used as ordinary subroutines during the course of the program.

----------MEAS device----------

Refer to description of the specific device being addressed by this statement (ACZ, DCM, DCS, etc).

----------NAILBIND----------

The statement DECLARE NAILBIND can used for analog testing on nails that belong to the SpeedPlus subsystem. Refer to DECLARE NAILBIND.

----------NAILOFFSET----------

General

This command is used for conveniently relocating a pin list. Following a NAILOFFSET command, all subsequent statements will have the specified offset number added to each of the pin numbers in their pin list. The offset number must be a multiple of 16 for a 2270 or 2271, or a multiple of 128 for a 2272. This corresponds to the number of pins supported by each scanner or driver/sensor board.

If the SpeedPlus option is used, FNAILOFFSET provides the offset for fast nails (but does not affect normal nails). Similarly, NAILOFFSET works on normal test nails, but does not affect fast nails.

Format

NAILOFFSET (offset); or FNAILOFFSET (offset);

where:	NAILOFFSET or FNAILOFFSET	Command keyword.
	offset	The value to be added to each pin in subsequent pin lists. For NAILOFFSET, the value must be a multiple of 16. May be either a floating-point constant or variable. For FNAILOFFSET, the value must be a multiple of 64.

Example

1. LET NOFF=0;
   WHILE NOFF<256 DO
   [NAILOFFSET (NOFF);
   TEST SHORTS AT (1-16) [WRITE 'SHORT AT BOARD NUMBER %G%' (NOFF/16+1);];
   LET NOFF=NOFF+16;];

   Meaning: Initialize nail offset variable (NOFF) to 0 and set up a loop to check shorts for each of 16 consecutive scanner boards (nails 1-256).

   After each board is tested, increment NOFF by 16 and offset the next scanner pin list by this value, using the NAILOFFSET statement. Use the value of NOFF to identify any board at which a short is detected.

Rules and Considerations

1. Scanner and driver/sensor pins can only be offset by multiples of 16 on the 2270, 2271, and 2275, and by 128 on the 2272.

2. NAILOFFSET affects the pin list in all subsequent statements having pin lists.

3. Once defined, an offset will continue to affect subsequent pin lists until specifically changed by another NAILOFFSET statement.

4. When NAILOFFSET is used on a 2271 or 2272, nail assignment software will only generate a single wiring list for the original (non-offset) nails. The user must then transpose the nail assignment results for successive boards.

--------ON ABORT--------

General

This statement can be used to define shutdown functions which the system will automatically perform upon detecting an abort condition. An abort condition may result from:

1. Any fatal Run-Time system error (hardware or software),
2. Pressing CTRL C on the keyboard (or the RESET button on the keypad),
3. Other abnormal program exits.

Format

ON ABORT CALL subroutine-name( );

or

ON ABORT EXIT;

where:	ON ABORT	Command keywords.
	CALL subroutine-name( );	CALL statement causes transfer of program control to the indicated subroutine.
	EXIT	EXIT statement stops further device testing, initializes all test hardware and restores default system behavior.

Examples

```
1.  DECLARE SHUTDOWN( );
            '
            '
    ON ABORT CALL SHUTDOWN( );
            '
            '
    END;
    SUBROUTINE SHUTDOWN( );
    SET UPS(4) V=0 I=0.5;
    DELAY 20M;   /*Wait for capacitors*/
    CLEAR UPS(4);    /*Disconnect power suppply*/
    SET PIO(0) HRLY(OPEN 8);    /*Open system isolation relay*/
    END SUBROUTINE SHUTDOWN;
```

Meaning: If the system detects an abort condition, subroutine SHUTDOWN will be called, turning off UPS(4) and disconnecting it from the system.

Rules and Considerations

1. An ON ABORT statement is not local to the main program or to a subroutine.

2. The subroutine must have a null argument list.

3. The execution of a new ON ABORT statement supersedes the effects of all previously executed ON ABORT statements.

4. Once an abort condition transfers program control to an abort subroutine, that routine can then issue a new ON ABORT statement which the system will honor if another abort condition occurs.

5. In the case of multiple abort conditions and subroutines, abort processing terminates at the completion of the currently active subroutine (i.e., program control never returns to the calling routine).

----------OPEN----------

General

The OPEN statement establishes a channel over which data can be transferred to and from a file or device via READ and WRITE statements. The OPEN statement assigns a user-selectable name to the channel, identifies the type of I/O operation (read, write or append) and specifies the device or file associated with the subsequent data transfers.

The keyboard/keypad and display are always implicitly OPEN and require no OPEN statement. Also, the special files LOGFILE, MESFILE, and I488 (if the I488 option is present) are always open. Two additional channels may be explicitly assigned to other files and devices, for a maximum of 4 active channels.

Format

```
OPEN ID=identifier DEV='file-spec' MODE='mode';
```

where:

OPEN	Command keyword.
ID=identifier	Assigns a channel name to the specified device or file. That name must then be used in all subsequent I/O activity with that device or channel. ID is the keyword; identifier is the user-assigned channel name.
DEV='file-spec'	Specifies the device or file being OPENed for the subsequent I/O data transfers. DEV is the keyword. The file-spec must be enclosed in single quote marks ('), and consists of either the user-assigned file name or the system-defined device keywords (i.e., TI:, LPO:, etc, as described in Appendix A). The keyboard and the display are implicitly opened at the start of each test program and need not be specified in an OPEN statement.
MODE='mode'	Defines the type of I/O transfer that will take place. READ indicates that data can only be read from the file or device, WRITE indicates that data can only be written into a new file or on a device and APPEND indicates that data can be appended to an existing file.

Examples

1. OPEN ID=APPLE DEV='LB1:TESTPROG1.A' MODE='READ';

   Meaning: Open file LB1:TESTPROG1.A for subsequent read access and assign that file to a channel identified as APPLE.

2. OPEN ID=PEAR DEV='LPO:' MODE='WRITE';

   Meaning: Attach the strip printer (LPO:) for subsequent write access and assign it to a channel named PEAR.

3. OPEN ID=GRAPE DEV='LB1:MORE.ADD' MODE='APPEND';

   Meaning: FIle LB1:MORE.ADD is assigned to a channel named GRAPE to permit subsequent WRITE ID=GRAPE statements to add data to the end of the file.

Rules and Considerations

1. The file-spec in DEV='file-spec' and mode in MODE='mode' must be enclosed in single quote (') marks.

2. Be sure to include the colon (:) after the device name (e.g., LPO:), otherwise the compiler will assume that a file name is being specified (e.g., file LPO).

3. The keyboard/keypad and display are implicitly OPENed at the start of a test program. In addition, up to 2 other channels may be opened explicitly via OPEN statements for a maximum of 4 OPENed channels. Note, however, that, if the keyboard and display were explicitly OPENed and assigned to specified channels, no other channel assignments could be made.

4. It is good practice to CLOSE explicitly OPENed channels, before the end of the test program (see CLOSE statement).

5. A channel must be OPENed to a file or device (other than the keyboard and display) prior to addressing a READ or WRITE statement to that file or device. It is recommended that all OPEN statements appear at the beginning of the test program (following the DECLAREs). In this way, if for any reason the specified channel cannot be opened, the user will be forewarned by an error message before the test program is executed.

6. Any file variable assigned by an OPEN statement must be DECLAREd at the start of the program.

7. ID=LOGFILE (refer to Data Logging in the System Operation manual) and ID=MESFILE (refer to Section 5 of this manual) should not be explicitly OPENed.

----------PAUSE----------

General

The PAUSE statement temporarily suspends program execution while the user performs some manual operation. Execution is resumed when the START/CONTINUE button on the keypad is pressed.

Format

```
PAUSE;
```

Example

1.
```
WRITE 'PRESS CONTINUE BUTTON WHEN READY';
PAUSE;
TEST DCV . . . . . ;
```

Meaning: The WRITE message is displayed and the program temporarily halted. When the user presses START/CONTINUE, the TEST DCV statement is executed.

Rules and Considerations

1. PAUSE statements can contain only the keyword PAUSE and no other parameters.

-------PEXCLOSE, PEXGND, PEXOPEN--------

These statements allow user control of the 2272 pin expander relays, and are for use in 2272 OVP programming only. Refer to para 7.5 in the 2272 System Service manual (supplied in the service kit) for more information.

----------READ----------

General

Once a channel has been OPENed, a READ command will direct data from that channel to the variable(s) indicated in the statement. If no channel has been specified, the keyboard is assumed to be the input device.

In addition to reading data from an input file or device, the READ statement can also be used to convert ASCII data to a numeric format. This feature, which is useful in manipulating data for the I488 Bus option, is demonstrated in Example 4 and described in the Rules and Considerations section. Refer to para 6.3, I488, for use of READ with IEEE-488 bus instruments.

Format

```
READ ID=channel 'format-characters' parameters-list [error-command;];
```

where:

READ — Command keyword.

ID=channel — Identifies the channel through which the data transfer will take place. ID is a keyword; channel is a user-assigned name previously defined in an OPEN statement. If ID=channel is omitted from the statement, the READ statement is directed to the keyboard/keypad. The channel I488(n) is pre-defined and does not need to be OPENed.

'format characters' — These characters specify the format of the input data. Each character must be preceded and followed by a % sign (e.g., %S%) and the list of format characters must be enclosed between single quote marks ('). Each format character defines the format for the corresponding variable in the parameters-list (i.e., the first format character is associated with the first variable, the second with the second variable, etc). The following format characters may be used in a READ statement:

- **%S%** Read a character string up to the first white space (e.g., space, tab, etc). Typically, this corresponds to a single character or word.

- **%R%** Read a record (i.e., a line of data). This format is useful in examining individual lines of data in a file or in reading a full line of data from the keyboard. A record typically consists of a line of data terminated by a line feed character (octal 012). Note that the RETURN key on the keyboard generates both a carriage return and line feed code, thus terminating a record at the end of the typed line. Non-typical terminations of a record include reaching the end of a file prior to a line feed character and DECLAREing a character string variable shorter than the actual record length that is read into it. The terminating character

'format characters' (cont)	(i.e., line feed or end of file) is not stored in the READ variable.
	%F% Read a number in either floating point or exponential (%E%) notation.
parameters-list	Lists the variable(s) in which the input data will be stored. The format for each variable is defined by the associated format character. Parameters in this list are separated by commas.
[error-command;]	If the READ statement cannot be executed or if an end-of-file is detected, this error command is executed.

Examples

1.
```
DECLARE CSTRING RESP (20);
      '
      '

READ '%S%' RESP;
```

Meaning: Read a word (i.e., a character string up to the first white space) from the keyboard and assign it to variable RESP. Since no ID= is specified, the keyboard is assumed as the input device. For this example, if the user typed YES and pressed RETURN, the character string YES would be assigned to the variable RESP. If the user typed YES MAYBE and pressed RETURN, only the YES would be accepted since YES is followed by white space.

2.
```
DECLARE CSTRING MSG (80);
DECLARE VAL;
OPEN ID=PEACH DEV='SC:FOO.BAR' MODE='READ';
      '
      '

READ ID=PEACH '%S% %F%' MSG,VAL;
```

Meaning: Transfer the first word in file SC:FOO.BAR into variable MSG. After encountering the first white space in the message, transfer the floating point number into variable VAL.

3.
```
DECLARE CSTRING LINE (80);
OPEN ID=FILEABC  DEV='SC:MSG.A'  MODE='READ';
      '
      '

READ ID=FILEABC '%R%' LINE;
```

Meaning: Read a record from file SC:MSG.A into character string variable LINE. Terminate the record when a line feed code (octal 012) is received.

4.
```
DECLARE CSTRING INASC(3);
DECLARE NUM;
      '
      '

LET INASC='490';
READ ID=INASC '%F%' NUM;
```

Meaning: ASCII character string '490' is assigned to variable INASC. This character string consists of octal codes: 064 071 060. The READ statement converts this ASCII character string to the floating-point number 490 and stores it as variable NUM.

Rules and Considerations

1. The format-character field must be enclosed by single-quote marks (').

2. To read data from a file or device other than the keyboard, an OPEN statement directed to that file or device must precede the READ statement, and the READ statement must be directed to that OPENed channel.

3. When the keyboard is specified as the input device, no internal action occurs until the RETURN key is pressed. This enables the user to edit the data and correct errors before executing the READ command. During a READ from the keyboard, the following keys are available for editing:

   CTRL U - Delete the whole line of type;

   DELETE - Delete the last character typed;

   RESET or CTRL C - Abort the input.

4. In various programming applications, numeric data are represented in ASCII format as, for example, data received over the optional IEEE-488 bus. In order to perform arithmetic operations on these ASCII data, they must first be converted to numeric format. For example, the ASCII character string representation of the number 490 consists of the following 3 bytes (in octal): 064 071 060. Before any arithmetic operations can be performed on this number, it must be converted to a binary representation. As shown in Example 4, a READ statement can be used to convert from one format to the other. Data read from the IEEE-488 bus can be converted directly, as follows:

```
READ ID=I488(n) '%F%' variable-name;
```

where:	READ	Command keyword.
	ID=I488	Identifies the source of the ASCII character string to be converted to numeric format (reads from the present talker on the IEEE-488 bus). The IEEE-488 bus does not need to be OPENed.
	n	The address of the IEEE device to read.
	%F%	Indicates the numeric form of the converted character string. %F% indicates floating point.
	variable-name	Name of the variable into which the numeric value will be stored.

----------RETURN----------

General

The RETURN statement may be used within a subroutine (either burst or non-burst type) to cause an immediate exit from that subroutine. Program control is automatically transferred to the statement following the CALL (for non-burst) or GOSUB (for burst).

Format

```
RETURN;
```

Example

```
1.      CALL RTEST(MAXR=NEWR);
            '
            '
        END;

        SUBROUTINE RTEST(FLOAT MAXR);
            '
            '
        IF RX<MAXR THEN BRANCH TEST2;
        RETURN;
 TEST2: TEST R . . . . ;
            '
            '
        END SUBROUTINE RTEST;
```

Meaning: If the value of RX is less than the value of MAXR, transfer to TEST2 and continue with the subroutine. If RX is equal to or greater than MAXR, exit the subroutine and return to the statement following CALL RTEST.

-------SCNCLOSE( ), SCNOPEN( )---------

These statements allow user control of the 2272 scanner relays, and are for use in 2272 OVP programming only. Refer to para 7.5 of the 2272 System Service manual (supplied in the service kit) for more information.

----------SDSOPT( )----------

This statement allows the user to simultaneously connect the driver/sensor and scanner systems to a given test nail (2271 only), and to control operation of the driver/sensor current sources. SDSOPT is normally for use in OVP programming. Please refer to section 7.5 of the 2270/2271 or 2272 System Service manual for more information.

----------SET device----------

Refer to description of the specific device being addressed by this statement (i.e., ACZ, DCM, DCS, etc).

----------SUBROUTINE/END SUBROUTINE----------

General

The SUBROUTINE and END SUBROUTINE statements are used to define the boundaries of a subroutine. The keyword SUBROUTINE must appear in the first line and END SUBROUTINE in the last line. Subroutines must be defined outside the main body of the program (after the END statement) and are accessed via the CALL statement (described earlier in this section.)

Internal subroutines are written in 227X test language and are included in the test program file following the END statement. External subroutines are written in a high-level language other than 227X test language and are assembled and installed in the system as separate tasks (see Appendix C). Both types of subroutines, however, are called in an identical manner via the CALL statement.

Format

```
SUBROUTINE  name (argument-list);
        '
        '
        '
statements
        '
        '
        '
END SUBROUTINE name;
```

where:

SUBROUTINE END SUBROUTINE	Command keywords.
name	User-assigned subroutine name. Optional when used in the END SUBROUTINE statement.
argument-list	List of DECLAREd variables used in the subroutine. Arguments must appear in the same order and in the same format as they appear in the DECLARE statement.
statements	All statements (including bursts) allowable in the main program are allowed in the subroutine.

Example

1. Main Program

```
 '
 '
CALL FAILR (PRINT=LOG, MVAL=RESULT);
 '
 '
END;

SUBROUTINE FAILR (FILE PRINT, FLOAT MVAL);
 '
 '
subroutine statements
 '
 '
END SUBROUTINE FAILR;
```

Meaning: Subroutine FAILR is defined immediately following the END statement in the test program.

Rules and Considerations

1. Internal subroutines must be defined outside the main program, i.e., following the END statement.

2. The argument list in the SUBROUTINE statement must be the same as in the DECLARE statement (i.e., same argument names, same order, etc)

3. Global variables should not be included in the argument list of a CALL or SUBROUTINE statement. It is a recommended practice, however, to re-DECLARE all global variables used in a subroutine at the beginning of that subroutine.

4. One subroutine may call another subroutine, in which case the DECLARE statement for the subroutine being called must appear in the body of the calling subroutine.

5. All local variables used within a subroutine must be DECLAREd in the body of that subroutine.

6. Even if no arguments are passed to it, the subroutine statement must include the parentheses (i.e., SUBROUTINE Name( )).

7. As with other statements an END SUBROUTINE statement may have multiple labels.

8. Subroutines cannot be used in a fast (SpeedPlus) burst.

----------SUBSTR (function)----------

General

A substring is a portion of a character string. For example, if a character string has been defined as MEASURED VALUE OF R25 IS 390 OHMS, then a substring could be defined as any portion of that character string (e.g., R25 IS 390 OHMS).

SUBSTR is a function that can be used to define a substring. It specifies the name of the character string from which this substring is to be taken, the position of the first character, and the number of characters in the substring.

Format

SUBSTR (char-string name, first-char-position, length);

where:	SUBSTR	Keyword used to define a substring.
	char-string-name	The name of a DECLAREd character string.
	first-char-position	An integer specifying the position in the character string from which the first substring character will be taken, where the first character in the string is specified as 1.
	length	The number of characters in the substring.

Example

```
1.  DECLARE CSTRING Y(20);
    DECLARE CSTRING X(30);
    LET X='THIS IS A CHARACTER STRING';
      '
      '
      '
    LET Y=SUBSTRING(X,11,16);
```

Meaning: The new substring Y will be: CHARACTER STRING.

Rules and Considerations

1. SUBSTR is a function and therefore cannot be used alone, but only as part of a statement.

2. Character string (or substring) variables must be DECLAREd before they are used.

3. If a SUBSTR argument is out of the string range, a suitable default value will be used (i.e., the value of the closest end of the string).

----------TEST device----------

Refer to description of the specific device being addressed by this statement (i.e., ACZ, DCM, DCS, etc).

----------TIME( )----------

General

The TIME( ) function is used to retrieve time information from the operating system. A floating point number is returned, specifying the number of seconds since midnight. This information can then be used to determine execution time of various program segments or of an entire program.

Format

TIME( )

Example

```
1.  LET T1 = TIME( );
      '
      '
    program segment
      '
      '
    LET T2=TIME( );
    LET DELT=T2-T1;
```

Meaning: At the start and at the end of a program segment, store the system time in floating-point variables T1 and T2, respectively. Then compute the interval between the start time and end time, and store the results in DELT. For example, if the start time were 36120.3 and the end time were 36152.7, then the time interval for executing that program segment would be approximately 36152.7 - 36120.3 = 32.4 seconds.

Rules and Considerations

1. TIME( ) is a function and thus cannot be used alone but only as part of a statement.

----------USE----------

General

This statement defines the default values for failure-reporting (diagnostic) options used in subsequent instrumentation statements (refer to para 5.3). Options set by a USE statement apply to subsequent instrumentation statements, except those statements with explicit parameters indicating otherwise.

Format

USE fail-options-list;

where: USE Command keyword.

fail-options-list Specifies default values for one or more test fail option(s), i.e., NOFSUB, FSUB=, PMSG=, CMSG=, DMSG=.

Example

1. USE FSUB=FAILSUB DMSG='NOMINAL';

   Meaning: Unless otherwise specified in a subsequent instrumentation statement, FAILSUB will be used as the FSUB subroutine and character string 'NOMINAL' will be used as diagnostic message DMSG whenever an instrumentation statement fails. These two default values will remain effective until respecified by a subsequent USE statement.

Rules and Considerations

1. The USE statement pertains to instrumentation statements lexically following it (i.e., those statements following USE in compile-time order, not necessarily run-time order).

2. Parameters within each instrumentation statement can override the default fail options defined by a USE statement. This overriding effect, however, applies only within the instrumentation statement containing the parameters.

3. When an instrumentation statement fails, options are selected according to the following priorities:

   1 - Those options explicitly specified in the FAILed statement
   2 - Those options defined by a previous USE statement
   3 - The initial defaults (nulls and zeros)

----------WHILE-DO----------

General

The WHILE-DO statement causes a program loop to be executed as long as the WHILE condition is true. The WHILE condition is evaluated at the start of each pass and, if satisfied, the loop statements are executed. The loop boundaries are defined by the WHILE-DO keywords in the first statement followed by the sequence of loop statements between brackets ([]).

Format

```
WHILE condition DO
[statements;];
```

where: WHILE-DO Command keywords

condition Specifies under what condition looping will continue. Looping continues as long as the condition is true. May be either an expression (e.g., VAR>20) or a Boolean variable that can take a true or false value.

[statements;] Brackets are used to define the boundaries of the loop. A left bracket ([) precedes the first statement, a right

[statements] precedes (1) follows the last statement. All statements
[,item] within one bracket must be separated by a semicolon
(;), including the last statement.

Example

```
1.  DEF FNA a ZZ
    WHILE FNA<>0 DO
    .
    .
    statements
    .
    .
    DEF FNA = FNA + 'ZZ
```

Warning: Beware the loop is never terminating for after each pass,

Rules and Considerations

1. Any entry into the loop is a test, and, if a FALSE result is encountered because the results are unpredictable and erroneous.

2. Enclose the statements within a right and a left bracket [ ], and terminate each statement with a semicolon.

Purpose

Used in conjunction with an OPEN statement for, like WRITE, to create, or open an existing, a data stream to be written to by the normal WRITE statement. The WRITE statement can direct a specific character string, or the value of given variables, or a combination of both, to the output stream. Values assigned in the WRITE statement cause the current or specified variable to be converted to the string in place of the expression.

As described in Rules and Considerations (item 8), certain code sequences also permit a WRITE statement to control the operation of the display, such as blanking, scrolling, moving the cursor, and returning the video control.

The WRITE statement provides a convenient method of converting numeric values to ASCII format. This feature, while useful for sending ASCII-coded data and commands to instruments on the IEEE-488 bus option, is described in more detail in the Rules and Considerations (item 7). Refer to Chapter 7 and the Rules and Considerations (item 7) before attempting to use the WRITE statement with IEEE-488 bus instruments.

Syntax

WRITE [#channel] 'format-string' parameter-list;

where WRITE Command keyword.

[#channel] Specifies the channel through which the data transfer will take place. If is a keyword; channel is a

user-assigned name previously defined in an OPEN statement. If ID=channel is omitted from the statement, the WRITE statement is directed to the display terminal. The channel I488(n) is pre-defined and does not need to be OPENed.

'format string' May consist of a character string, a group of %formatting-sequences% or a combination of the two, as shown in the following examples:

(1) Character String

If a format-string is simply composed of a character string as follows:

'RESISTANCE TEST FAILED'

then that character string is simply written as it appears between the single quotes (').

(2) Formating Sequences

Format strings may also consist of one or more formating sequences (i.e., %S%, %F%, etc). These sequences may be used either to indicate the presence of special ASCII characters in the string or to specify the data format for variables in the parameters list.

The following data formatting sequences are allowed in the format-string of a WRITE statement:

%G% - Will print 1 to 3 digits to the left of the decimal point and use scaling factors MEG, K, M, U, N and P (e.g., 10.5 K). Exponents are used for numbers greater than 10E+6 or less than 10E-12.

%E% - Same as %G% except that the exponential form is always used rather than scaling factors (e.g., 10.5E3).

%F% - Floating-point number (e.g., 3.14159). Numbers greater than 32767.4 (largest integer) are printed in %E% format. Numbers too small for %F% format are printed as 0.0.

%S% - Character string (e.g., TEST 1).

%R% - Equivalent to the %R% format in READ. Will write a line of data (i.e., character string), followed by a %NL%.

The %E%, %F% and %G% formats may also specify the minimum field width and the precision of the number. Item 5 in Rules and Considerations further describes the %E%, %F%, and %G% formats and gives examples illustrating their use.

'format string' (cont) In addition to specifying data formats, the % symbols may be used to specify non-printing ASCII characters in the format string. For example, inserting a 3-digit octal code between % signs (e.g., %011%) inserts that ASCII function (e.g., horizontal tab) at that position in the character string. The mnemonic representations %NL%, %CR%, and %LF% may be used in a format string to signify New Line, Carriage Return, and Line Feed, respectively.

(3) Character Strings with Formating Sequences

A format string may also consist of a character string with embedded %formating-sequences%. Each embedded formating sequence causes either a special ASCII code or the value of a specified variable to replace that sequence in the string. For example, the following format string in a WRITE statement:

```
WRITE '%S% FAILED %NL% MEASURED VALUE = %G%' A, B;
```

would cause the value of character string variable A to replace %S%, the value of variable B to replace %G%, and a carriage return/line feed to be inserted between FAILED and MEASURED.

parameter-list Optional. A list of variables to be written by this statement. The format of each variable is determined by %formating-sequences% in the string, with the first sequence defining the format of the first variable. Variables in the list are separated by commas.

Examples

1.
```
WRITE '%F%' VAR;
```

Meaning: Display the value of variable VAR, in floating-point format. Note that, since ID=channel is not specified, the display terminal is assumed to be output device.

2.
```
WRITE 'SHORTS TEST FAILED';
```

Meaning: Display the message SHORTS TEST FAILED.

3.
```
OPEN ID=LOG   DEV='LPO:'  MODE='WRITE'; WRITE ID=LOG '%S% FAILED.
MEASURED GAIN=%F% %NL%' NAME, BETA;
```

Meaning: Open a channel identified as LOG for subsequent WRITE outputs to the strip printer (LPO). If the character string variable NAME has been defined as Q25 and the floating-point variable BETA as 100, the following message will be sent to the printer.

```
Q25 FAILED.  MEASURED GAIN=100
```

The New Line code %NL% then causes the message to be printed.

4. WRITE 'RESISTOR R%F% MEASURED %G% OHMS' RNUM,RVAL;

   Meaning: If RNUM=10. and RVAL=2.8K, the following message would be displayed:

```
RESISTOR R10 MEASURED 2.8K OHMS
```

5. OPEN ID=ORANGE DEV='LB1:FILEA.B' MODE='APPEND';
   WRITE ID=ORANGE 'APPEND THIS STRING TO THE FILE%NL%';

   Meaning: Open existing file LB1:FILEA.B and append character string 'APPEND THIS STRING TO THE FILE' to it.

6. OPEN ID=LOG DEV='LPO:' MODE='WRITE'; WRITE ID=LOG '%NL% TEST RESULTS%NL%';

   Meaning: Send a CR code and LF code (new line) to the strip printer along with the character string TEST RESULTS. The %NL% then causes the printer to print TEST RESULTS and advance paper to the next line.

7. WRITE 'THIS %LF% AND THIS %012% ARE BOTH LINE FEED CODES';

   Meaning: The following message will be displayed on the CRT.

```
THIS
    AND THIS
            ARE BOTH LINE FEED CODES
```

8.
```
DECLARE NUM;
DECLARE CSTRING OUTASC (10);
     '
     '
LET NUM = 490.
WRITE ID=OUTASC '%F%' NUM;
```

   Meaning: Floating-point variable NUM contains the numeric value 490. The WRITE statement converts this number to an ASCII character-string 490 (i.e., octal 064 071 060) and stores it in string variable OUTASC.

Rules and Considerations

1. The format string in a WRITE statement must be enclosed between single-quote marks ('). Also, a literal character string (i.e., a character string enclosed by single quote marks) cannot exceed 80 characters.

2. If data are to be written on any device other than the display, an OPEN statement directed to that file or device must precede the WRITE statement and the WRITE statement must contain an ID= addressed to that OPENed channel.

3. A file must have been previously created before data can be appended to it. When a file is opened for an append operation, the version number of the file does not change.

4. Any WRITE statements directing messages to the Centronics Model P1 Microprinter require a %NL% carriage return/line feed at the end of the string to indicate an end-of-line to the printer. Otherwise the line will not print.

5. A number presented in scientific notation (%E%) or floating-point (%F% or %G%) representation can be formatted with the following qualifications between the % symbols, as follows:

E-format:
%-w.d E% (e.g., %6.5E%)

F-format:
%-w.d F% (e.g., %-5.3F%)

G-format:
%-w.d G% (e.g., %5G%)

where:

-	Optional. Specifies left justification of the number.
w	Minimum field width. The number will be presented in a field at least this wide, and wider if appropriate. If the number has fewer digits than specified by the field width, it is padded with as many spaces as necessary, or with a leading zero if w has a leading zero.
d	The number of digits printed to the right of the decimal point. Default value is 6 minus the number of digits to left of the decimal point.
E, F or G	Denotes E, F or G format.

The following examples illustrate the use of %E%, %F% and %G% formats.

Value	%G%	%10.4G%	%5G%	%F%	%6.2F%	%E%
637.852	637.852	637.8520	637.9	637.852	637.85	637.852E+0
10,361.47	10.3615K	10.3615K	10.36K	10361.5	10361.50	10.3615E+3
0.3874	387.4M	387.4000M	387.4M	0.3874	0.39	387.4E-3
2.3P	2.3P	2.3000P	2.3P	0	0.00	2.3E-12
11.67E+18	11.67E+18	11.6700E+18	11.67E+18	11.67E+18	11.67E+18	11.67E+18

6. The % and '(single quote) are special characters in a character string. To use either of them as text symbols in a WRITE statement, they must appear as follows:

%% will write the percent symbol(%)

%'% will write the single quote mark(').

7. In various programming applications, the optional I488 Bus, internal numeric data must be converted to ASCII format. For example, suppose the floating-point decimal number 490 must be converted to its octal-coded ASCII representation, 064 071 060, and sent to an IEEE-488 instrument. This conversion can be made via a WRITE statement as follows:

WRITE ID=string-name '%F%' variable-name;

where: WRITE	Command keyword.
ID=string-name	Specifies an ASCII character string variable into which the converted numeric format will be stored.
'%F%'	Indicates that the corresponding variable-name represents a floating-point number.
variable-name	Name of the variable from which the numeric value is to be taken.

8. Through the use of special escape sequences, a WRITE statement can control the operation of the display and keypad indicators as shown in the following table. The ESC character must be represented in the WRITE statement by its ASCII octal code between % symbols (%033%).

NOTE

The escape code should be specified as %033%, i.e., the octal representation of the ASCII escape character. As an example, the statement WRITE '%033%:A' will erase the screen.

Escape Sequence	Function
'%033%:A'	Erase screen and clear all attributes
'%033%:B'	Underline (turn on line 8 of character dot matrix)
'%033%:C'	Underline (turn off scan line 8)
'%033%:D'	Underblink on. Any turned-on scan lines blink (independent of character)
'%033%:E'	Enable scroll
'%033%:F'	Disable scroll
'%033%:G'	Underblink off
'%033%:H'	Reserved (do not use)
'%033%:I'	Double width on
'%033%:J'	Double width off
'%033%:K'	Underline (turn on scan line 9 of dot matrix)
'%033%:L'	Underline (turn off scan line 9)
'%033%:M'	Intensified video on
'%033%:N'	Intensified video off
'%033%:O'	Reversed video on
'%033%:P'	Reversed video off
'%033%:Q'	Blink character on
'%033%:R'	Blink character off
'%033%:S'	Reserved (do not use)
'%033%:T<a1> <val2>'	Reserved (do not use)
'%033%:U'	Enable cursor
'%033%:V'	Disable cursor
'%033%A'	Cursor up one, no scroll
'%033%B'	Cursor down one, no scroll
'%033%C'	Cursor right one
'%033%D'	Cursor left one
'%033%H'	Cursor home
'%033%I'	Cursor up one, roll down
'%033%J'	Erase to end of screen
'%033%K'	Erase to end of line

'%033%Y%<line#>%%<column#>%' Direct cursor positioning where <line#>= 040 to 067 (octal) for rows 0-23 (decimal); and <column#>= 040 to 157 (octal) for columns 0-79 (decimal). An attempt to move the cursor beyond these limits will place it at the boundaries. Example:

```
'%033%Y%054%055%'
line 12__/   /
column 13___/
```

NOTE

As an exception to the above statement, do not use octal 045 (i.e., %045%) to specify line 6 or column 6. Since octal 045 is the ASCII code for a % symbol, the term %045% would be interpreted as 3 successive % symbols (see item 6 in Rules and Considerations). For octal 045 use %% to represent a single % character, instead of using the octal value; for example:

```
WRITE '%033%Y%%%040%';
                 /   /
  line 6 ---    /
    column 0 --
```

Keypad Indicators

'%033%%%A'	Left PASS light on
'%033%%%B'	Left PASS light off
'%033%%%C'	Left PAUSE light on
'%033%%%D'	Left PAUSE light off
'%033%%%E'	Left FAIL light on
'%033%%%F'	Left FAIL light off
'%033%&A'	Right PASS light on
'%033%&B'	Right PASS light off
'%033%&C'	Right PAUSE light on
'%033%&D'	Right PAUSE light off
'%033%&E'	Right FAIL light on
'%033%&F'	Right FAIL light off

9. Print size can be controlled on the Centronics P1 printer with 3 control codes. '%035%' sets the print size to 20 characters per inch (7.9 per cm); '%036%', to 10 characters per inch (3.95 per cm); and '%037%' to 5 characters per inch (1.97 per cm).

10. A WRITE ID=MESFILE or WRITE ID=LOGFILE statement stores the specified WRITE data in a system message file or system log file (defined by monitor option in DIAGNOSE mode). Channels MESFILE and LOGFILE should not be explicitly OPENed or CLOSEd.

11. Be sure to design messages which will not exceed 80 characters on a line or the excess characters will be lost when the message is printed on the strip printer.

6.3 ANALOG INSTRUMENTATION STATEMENTS

This section describes all the analog (non-digital) instruments available in this system and the test language statements used to control and monitor these instruments. For inclusion of analog test statements in digital tests (e.g., for testing digital-to-analog converters), refer to the discussion of the BURST statement in para 6.4.4.

----------ACZ (AC Impedance Measurement Unit)---------

General

The ACZ consists of an ac source and a phase-sensitive meter. Impedance or admittance can be measured by applying the ac source voltage across the unknown impedance and measuring the resulting in-phase and quadrature (90 degree) voltage and current. From these measurements the system calculates the desired parameters, using complex algebra.

The ACZ can be controlled by MEASure, TEST, or CLEAR statements. SET ACZ is not allowable because the setup and automatic measure or test are inseparable actions. Measured capacitance values are in farads; inductances are in henries.

Any of the following functions can be specified:

Function	Meaning
RS	Series Resistance (ohms)
CS	Series Capacitance (farads)
LS	Series Inductance (henries)
XS	Series Reactance (ohms)
RP	Parallel Resistance (ohms)
CP	Parallel Capacitance (farads)
LP	Parallel Inductance (henries)
XP	Parallel Reactance (ohms)
Z	Impedance Magnitude (ohms)
Q	Quality Factor
D	Dissipation Factor

NOTE

In typical test measurements, CP is used instead of CS except for large capacitors, where the reactance is less than 25 ohms. LS is normally used for typical test measurements.

Format

```
TEST/MEAS function ACZ (unit) parameters [error-command;];
```

and

```
CLEAR ACZ;
```

where: TEST/MEAS/CLEAR — Command keywords. Must appear first in the statement.

function — Specifies the function being tested or measured. May be any of the following: RS, CS, LS, XS, RP, CP, LP, XP, Z, Q or D.

ACZ	Optional keyword for AC Impedance Measurement module.
(unit)	Optional. Used to identify the ACZ if system contains more than one unit. Default unit is (1).
parameters	Allowable parameters for this type of statement are: V= , F= , DLY= , BIAS= , HIPREC, limits= , MAX= , FORCE, INTO variable, FAIL(n). A MEAS statement requires the INTO variable parameter to store the measured value and a TEST statement required limits. All other parameters are optional. See Parameter Definitions.

NOTE

For parameters V= , F= , and BIAS= , the value must be specified as a floating-point number that falls within 5% of the indicated value. For example, the 1.0-V source voltage level must be specified in the range of 0.95 V to 1.05 Vrms.

[error-command;]	Optional. May contain any test language statements, including digital bursts. This error-command is executed if the instrumentation statement fails.

Parameter Definitions

V=	AC source voltage in volts RMS. May be 0.2 V or 1.0 V. Default is 0.2 Vrms.
F=	AC source frequency in Hz. May be 100, 1000 or 10000 Hz. Default is 1 kHz.
DLY=	Set-up delay in seconds. Maximum value is 1.999. Default is 0.
BIAS=	DC Bias voltage for ac source (in VDC). May be -0.3 V, +0.3 V, -1.5 V or +1.5 V. Default is 0 V.
HIPREC	Keyword that lengthens the integration period of the ACZ to provide a more precise measurement. Starting with software release 6, this is the default (and only) condition, and thus the keyword HIPREC is no longer required.
limits=	May be expressed as a high limit (HI=), low limit (LO=), or both, or may be specified as a nominal value (VAL=) with a tolerance (TOL=).
MAX=	Specifies a floating-point value that defines the starting point for auto-ranging. When MAX= is used with ACZ statements, the specification is the highest value expected for RS, RP, LS, LP, XS, XP, and Z. MAX= is the minimum value expected for CP and CS measurements. When measuring Q, MAX= specifies the maximum L expected. When measuring D, MAX= specifies the minimum C expected.
FORCE	Keyword that disables auto-ranging. Cannot be used with MEAS R.

FAIL(n) Fail vector bit n.

INTO Variable Measurement results are stored in this variable.

AT( ) This parameter is identical to the AT( ) construct in a pin scanner statement and performs the same function. It connects the MUX output channels to the appropriate UUT pins.

NOTE

Prior to executing a TEST or MEAS ACZ statement, proper MUX connections to the ACZ must be explicitly made via the following statement:

```
SET MUX AT(1:2:3:10:1,2,3,11);
```

This connects the ac voltage source to CHA, the ac current measure unit to CHB, the reference (Guard) bus to CHC, and analog ground to CHD.

Examples

1. `TEST Z LO= 13.6K HI= 15.4K [BRANCH Z42BAD;];`

   Meaning: Using existing MUX and SCANner pin connections, test the impedance to determine if it falls between the specified limits. The branch is taken if the impedance is outside these limits. The default frequency (1 kHz) is used. Similarly, the ac level defaults to V= 0.2 V with no bias.

2. `MEAS CP AT(15:16) HIPREC INTO CAP [BRANCH ERR1;];`

   Meaning: Measure parallel capacitance between scanner pins 15 and 16. The source frequency is set to 1 kHz (default value) and the ac source output is set to +0.2 V (default) with no dc bias. High precision is used and the measured value is stored in the variable CAP. If for some reason the measurement cannot be made, the error branch is taken.

3. `MEAS CP F=100 BIAS=+1.5 V=1.0 INTO CP1 [BRANCH CAPERR;];`

   Meaning: This statement is similar to the previous one except the parameters are changed. None of the defaults are used. Note that, since bias was desired, the higher value (1.5 V) of bias must be specified because the higher ac output level (1.0 V) was selected.

4. `TEST CP ACZ F=10.K VAL=470.P TOL=10 AT(10:11:12:12) FAIL(3) [BRANCH H100;];`

   Meaning: Figure 6-1 illustrates a typical instrument configuration for making guarded capacitance measurements. The capacitance to be measured is connected across pin scanner pins 10 and 11. The TEST statement configures the scanner to set up the ac source and detector to these pins. The MUX channels are automatically connected through the default assignments.

   The source is configured across the capacitor between scanner pins 10 and 11 in the TEST statement by applying it through CHA to pin 10, referenced to analog ground on CHC and CHD. Both voltage across and current through the capacitance are measured as shown. Scanner pin 11 reflects analog ground and is the low side for the voltage measurement as well as the current measurement input.

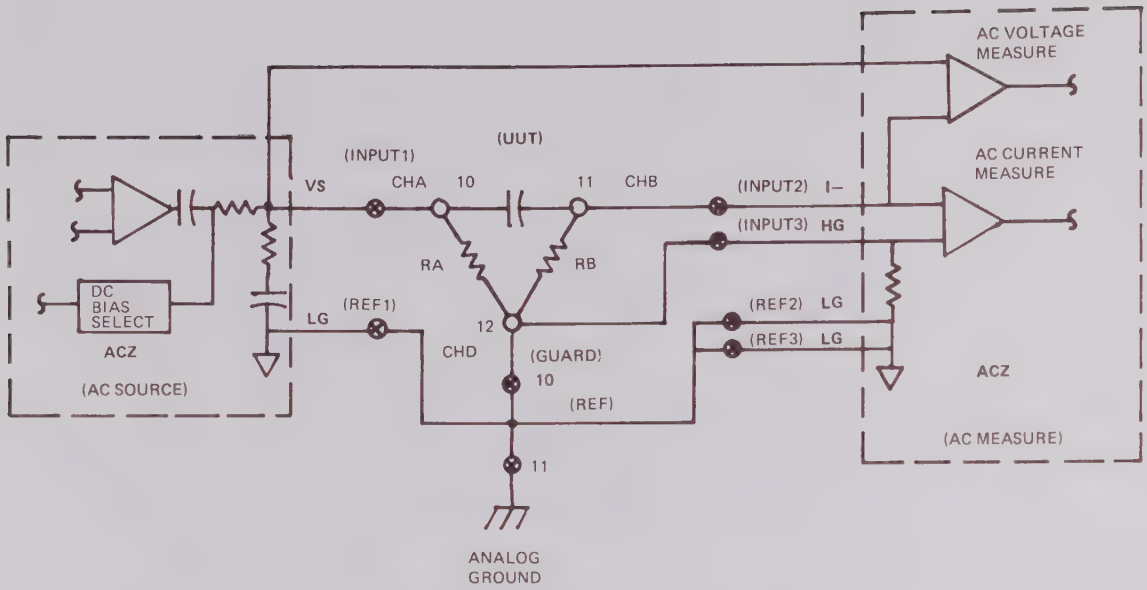


Figure 6-1. Capacitance Measurement (Parallel, Guarded).

Rules and Considerations

1. The device name ACZ need not be specified.

2. The function to be measured (RS, CS, etc) must be specified and must appear before the keyword ACZ (if used).

3. Be sure to connect the ACZ to the MUX before executing a TEST or MEAS ACZ statement (see Format). Otherwise, the program will attempt to use the existing MUX connections.

----------ARITH----------

General

The TEST ARITH statement is used primarily to test arithmetic quantities that are not directly measured by an instrument (e.g., the gain of a transistor, the average of some measured values, etc). This TEST ARITH statement has all the diagnostic capabilities of any other instrumentation statement, such as TEST DCM, MEAS ACZ, etc.

The statement compares an arithmetic expression with given limits. If the tested value falls outside the limits, the statement fails. If specified within the statement, a fail bit will be set, user diagnostics called, and an [error-command;] executed, the same as in any instrumentation statement.

Format

TEST ARITH arith-expression limits diag-option FAIL(n) [error-command;];

where: TEST ARITH — Command keywords.

arith-expression — The name assigned to the arithmetic expression being tested.

limits — Same types of limits as used in any other instrumentation statement:
VAL = nominal-value
TOL = tolerance
HI = high-limit
LO = low-limit

diag-option — Any of the user diagnostic options described in para 5.3 (e.g., FSUB=, PMSG=, etc).

FAIL(n) — Optional. Fail vector bit n. Set to 1 if the tested value falls outside the given limits. If n is a null (i.e., FAIL( )), then no fail bit is set. If FAIL(n) is not specified at all, then FAIL(1) is assumed.

[error-command;] — Optional. Executed upon failure, after return from user diagnostic routine.

Example

1. IF CBETA<0. THEN LET CBETA =-CBETA;
TEST ARITH CBETA LO=10 FAIL(5);

Meaning: The IF-THEN statement ensures that the value of CBETA will be positive. The TEST ARITH statement checks that value. If it is less than 10, fail bit 5 will be set.

----------DCM (DC Measurement Unit)----------

General

The DC Measurement Module (DCM) consists of a differential voltmeter and an ammeter. DC voltages from 0 V to ±128 V can be measured. The most sensitive range is 250 mV full scale. Current can be measured to ±160 mA, the most sensitive range being 9.76 uA full scale. The resolution in both modes is 13 bits plus sign. The DCM is protected against overvoltage up to 150 V and overcurrent up to 1 A.

NOTE

If the meter is overloaded, either because a range is forced or the stimulus is higher than the measurement capabilities, then a value of 100 A is reported for a current measurement. This forces a fail condition.

Ranges are not user-selectable. A recommended practice for efficient testing is to specify the highest acceptable tolerance limit of a measurement, using MAX=. This minimizes ranging, thereby hastening statement execution.

Format

SET/TEST/MEAS DCI/DCV DCM(unit) parameters [error-command;];

and

CLEAR DCI/DCV DCM(unit);

where: SET/TEST/MEAS/CLEAR Command keywords.

DCI/DCV	Selects either the ammeter (DCI) or voltmeter (DCV) function of the DCM. Must precede keyword DCM.
DCM	Optional. Keyword for DC Measurement Module.
(unit)	Optional. Used to identify the DCM if system contains more than one. Default unit is (1).
parameters	Allowable parameters for this type of statement are DLY=, RDLY=, limits=, MAX=, FORCE, INTO variable and FAIL(n). A MEAS statement requires the INTO variable parameter to store the measured value and a TEST statement requires limits. All other parameters are optional. See Parameter Definitions.
[error-command;]	Optional. May contain any test language statement (other than burst-only command). This error command is executed if the instrumentation statement fails.

Parameter Definitions

DLY=	Set-up delay in seconds. Maximum value is 1.999 seconds. Default value is 0.
RDLY=	Specifies delay (in seconds) between range changes. Maximum value is 1.999 seconds. Default value is 0.
limits=	May be expressed as a high limit (HI=), low limit (LO=), or both, or may be specified as a nominal value (VAL=) with atolerance (TOL=).
MAX=	Specifies a floating-point value that defines the starting-point for auto-ranging.
FORCE	Keyword that disables auto-ranging.
INTO variable	Measurement results are stored in the variable specified by this parameter.
FAIL(n)	Optional. Fail vector bit n. Set to 1 if a failure is detected. If n is a null (i.e., FAIL( )), then no fail bit is set. If FAIL(n) is not specified at all, then FAIL(1) is assumed.

Examples

1. SET DCV MAX=6.5 FORCE [BRANCH ERR1;];

   Meaning: Configure the DCM for a voltage measurement. The highest value the measurement is allowed to have without error is 6.5 V. The DCM is forced to use the most sensitive range that accommodates 6.5 V. If the setup cannot be made, the error branch is taken.

2. MEAS DCV INTO V3 [BRANCH E100;];

   Meaning: Measure the voltage and store the value in variable V3.

3. TEST DCI DCM MAX=.028 VAL=0.025 TOL=10 RDLY=.100 FAIL(14) [BRANCH E12;];

   Meaning: Test the current, allowing a 100-ms delay between each range change. The value is expected to be within 10% of 2.5 mA. If the tested value falls outside these limits, FAIL bit 14 is set, the error branch is taken, and the FAIL indicator is turned on.

4. CLEAR DCM;

   Initialize both the ammeter and voltmeter.

Rules and Considerations

1. The function DCI or DCV must be specified and must appear before DCM (if used).

2. The device keyword DCM is optional in the statement.

3. Ranges are not user-selectable, but specifying the MAX= parameter speeds up the internal range selection.

----------DCS (DC Source)----------

General

The DC Source can be used both as a voltage source (DCV) and as a current source (DCI) and can be controlled by a SET, TEST or CLEAR command.

The voltage source (DCV) is programmable between -16 and +16 V via the V= parameter. The I= parameter permits output current limiting to either of 2 values: 20 mA or 128 mA. If not explicitly specified in a statement, a 128-mA current limit is assumed.

The current source (DCI) is programmable between -128 mA and +128 mA via the I= parameter. The output voltage limit (compliance voltage) is fixed at 16 V. When the load resistance is high enough to develop an IR drop of 16 V across it, compliance has been reached, and the source is no longer linear beyond this point. Thus, the current level should be selected so that compliance is not reached.

NOTE

No [error-command;] option is available for DCS error branching after a SET statement. Error conditions in the DCS (such as excessive current) can, however, be checked with the statements of the form TEST DCI DCS [BRANCH error;] or TEST DCV DCS [BRANCH error;]. Refer to para 5.2.1.

Format

SET/TEST DCI/DCV DCS(unit) parameters;

and

CLEAR DCI/DCV/null DCS(unit);

where:

SET/TEST/CLEAR	Command keywords.
DCI/DCV	Keyword specifying the function being set or tested. Must appear before DCS.
DCS	Keyword for DC Source.
(unit)	Optional. Used to identify the DCS if the system contains more than one unit. Default unit is (1).
parameters	Used with SET or TEST only. V= must be specified for a voltage source and I= for a current source. All other parameters are optional. Allowable parameters for DCS are: V=, I=, DLY=, OFF (for DCI only), INTO variable and FAIL(n). See Parameter Definitions.

Parameter Definitions

V=	Voltage specification for the DCV source, programmable from -16 V to +16 Vdc (no default).
I=	Output current specification for the DCI source, programmable from -128 to +128 mA (no default). Also can be used to current-limit the DCV source to 20 mA or 128 mA (default value is 128 mA).
DLY=	Set-up delay in seconds. Maximum value is 1.999 seconds. Default value is 0.
OFF	Keyword that disables the DCI output (shorts high and low DCI outputs together), but still allows the current source to be set-up to its specified value. Only valid with the DCI source.
FAIL(n)	Optional. Fail vector bit n. Set to 1 if a failure is detected. If n is a null (i.e., FAIL( )), then no fail bit is set. If FAIL(n) is not specified at all, then FAIL(1) is assumed.

Examples

1. SET DCV DCS V=6.5;

   Meaning: The DCS is configured as a voltage source at a level of 6.5 V. Since it was not otherwise specified, current limiting is 128 mA.

2. SET DCV DCS V=5.0 I=20.M;

   Meaning: The DCS is configured as a voltage source at a level of 5.0 V. Current is limited to 20 mA.

3. SET DCI DCS I=75.M OFF;

   Meaning: The DCS is configured as a current source at a level of 75 mA but is NOT turned on.

4. SET DCI DCS I=10.M;

   Meaning: The DCS is set to 10 mA.

5. TEST DCI DCS;

   Meaning: The DCS is turned on in the current mode and a check is made to ensure that the compliance voltage limit has not been exceeded. If it has, the FAIL(1) bit is raised and, at the end of the program, the FAIL indicator is turned on.

6. TEST DCV DCS;

   Meaning: The DCS is turned on in the voltage mode and a check is made to ensure that the device is not in current limiting. If it is, the FAIL(1) bit is raised, and the FAIL indicator is turned on.

7. CLEAR DCV DCS(2);

   Meaning: The second of 2 voltage sources is initialized.

8. CLEAR DCI DCS;

   Meaning: The current source only is initialized.

--------HCS (High-Current Supply)--------

General

The High-Current Supply (HCS) is a program-controlled unit with voltage levels and current limits specified by the program. There are two types of HCS available as options, a 0 to 7 V @ 17 A unit and a 0 to 5 V @ 35 A unit. The 2270 can have one 17-A HCS installed; the 2271 can have up to 2 units and the 2272 up to 4 units of either or both types. Each HCS is separately programmed for voltage output and current limit by a dedicated optional dc source (see listing below). As an integral part of the unit, 3 relays, energized by a relay driver (listed below) on the system relay driver board PIO(0), connect the HCS outputs and sense lines to the UUT.

Connection via PIO(0):	Power (all)	HCS(1) RLY(16)	HCS(2) RLY(15)	HCS(3) RLY(14)	HCS(4) RLY(13)
Voltage/Current set:	2270	DCS(4)	--	--	--
	2271	DCS(3)	DCS(4)		
	2272	DCS(3)	DCS(4)	DCS(5)	DCS(6)

All HCS programming routines must include appropriate delays to allow for supply settling. Refer to "Programming Power and Ground Connections" in Section 4 of the Test Set Preparation manual.

The HCS output is not bipolar. The actual polarity of the voltage to the UUT depends on the connection of the HI and LO output leads to the test fixture and the configuration of jumpers JM1 - JM4. These are located on receiver block 2 of the 2270 and 2271 and on position 0 groups 1 thru 6, 8, and 9 of the 2272 test head. Refer to the Test Fixture manual for more information.

Format

```
SET DCV DCS(n) V=voltage;
SET DCI DCS(n) I=current;
TEST DCI/DCV DCS(n) [error-command];
```

where:	SET/TEST	Command keyword.
	DCS(n)	Optional DC source used to control the HCS.
	DCV	Keyword specifying the DCS(n) voltage source.
	V=voltage	Output voltage of the HCS (and DCS(n)).
	DCI	Keyword specifying the DCS(n) current source.
	I=current	Current limit of the HCS scaled down by a factor of -0.0001 (i.e., -100uA/A). Must be a negative value. See Example 1.

Example

1.
```
SET DCV DCS(4)  V=5;
SET DCI DCS(4)  I=-100U;
TEST DCI DCS(4) [WRITE 'HCS COMPLIANCE'];
```

Meaning: Set the HCS(2) output voltage to +5 V, with a 1 A current limit. Display the message if HCS(2) output voltage reaches compliance.

--------I488 (IEEE-488 Instrumentation Bus)--------

General

The IEEE-488 Bus option is GenRad's implementation of IEEE-488-1978.[1] With the IEEE-488 Bus option, up to 9 external instruments conforming to the standard can be attached to the test system and remotely operated using high-level test language programming. In operation, the test system required to operate a particular device attached to the bus depends on the operating

[1]Copies of IEEE STD 488 1978 Digital Interface for Progammable Instrumentation may be purchased from The Institute of Electrical and Electronic Engineers, Service Center, Department PB-8, 445 Hoes Lane, Piscataway, NJ 08854.

characteristics of that device, and it is therefore necessary to consult the remote-programming instructions supplied with the device. This section outlines basic programming considerations and provides an example of how the test language is used to control and monitor devices on the IEEE-488 bus.

The hardware required for this option consists of an interface module that plugs into the I/O electronics area of the 2270 2271, or 2272; or into the Electronics Cage of the 2275. Adequate bus cables are also needed to connect from the interface module to each instrument. The test system considers the IEEE-488 bus (referred to as IEEE-488 bus in this description) as another system device that can be controlled by the same type of instrumentation statements as any other system device (i.e., CLEAR, TEST, READ, WRITE, SET and MEAS). The run-time system handles the low-level bus protocols. Internal connections from the 9 BNC connectors on the rear panel of the Processor Bay (Electronics Bay in the 2275) to the instrument multiplexer MUX(2) are shown in the MUX description later in this section. Each IEEE-488 instrument used in the system plugs into one or more of these connectors. The IEEE-488 instruments can therefore be connected to test nails in the same way as the 227X's internal analog instruments.

A typical measurement on the IEEE-488 bus starts by connecting the IEEE-488 instrument to the appropriate test nails, using SET MUX and SET SCAN. The measurement is carried out with a command to initialize the bus to a known state (CLEAR I488), a second command to program an instrument to supply a signal or perform a measurement (WRITE ID=I488), and further commands to retrieve the results (READ ID=I488, TEST I488). The specific form of a transaction depends on the nature of the test and the programming requirements of the instruments being used.

The keyword I488 is used to direct the standard functions CLEAR, TEST, READ, and WRITE to the IEEE-488 interface. (The functions SET and MEAS can also be directed to the IEEE-488 bus, but they are generally less convenient than READ and WRITE.) The I488 statements perform 3 basic functions:

1. Interface Control. These commands are used to control and monitor the state of the IEEE-488 bus interface and all IEEE devices attached to it. They are issued via the following test language statements:

CLEAR I488;
Activates the Interface Clear line on the IEEE-488 bus. The specific response of an instrument depends on how the clear function is implemented.

TEST I488 MAXTIME=t line-parameter [error-statements;];
Waits t seconds for a request to appear on the management line specified by line-parameter, which can be either the I488 SRQ or the EOI line. If a request does not appear within the time-out, the error-statements are executed. The TEST statement can be used to determine, for example, whether a measurement has been completed.

SET I488 bus-commands;
Using standard IEEE-488 mnemonics, the SET I488 statement sends standard bus commands -- such as UNL (unlisten), GTL (go to local) -- to the bus. Refer to Table 6-2 for a list of valid commands.

2. Message Outputs (to the device). The WRITE statement is used to issue specific setup messages to a device. The message is in the form of a character string unique to the device being "talked to" and is issued via the following statement type:

```
WRITE ID=I488(n) 'format-string' parameter-list [error-statements;];
```

The parameter n gives the address of the instrument on the bus; the WRITE statement automatically issues the commands required to make the instrument a listener. WRITE then sends format-string to the instrument. The format-string is typically a series of ASCII characters that command an instrument to perform some action. It can include any combination of literals (the actual characters appearing between the single quotes) and variables (e.g., %S%), as in other WRITE statements. If the instrument does not respond properly to the commands, the error-statements are executed.

3. Data Input (from the device). Information placed by the addressed device on the IEEE-488 bus is read via the following statement type:

```
READ ID=I488(n) LENGTH=m
 'format-characters' variable-list;
```

When data are expected from an IEEE-488 instrument, the READ statement first places the instrument at address n in 'talk' mode. It then transfers a stream of characters from the instrument into variables named in the variable-list. READ can break the character stream into records or words, or convert strings of ASCII numbers to 277X floating-point numbers, as determined by the format-characters.

For general information on the READ and WRITE commands (in particular, the use of variables with format-characters and format-strings), refer to the descriptions of READ and WRITE in para 6.2. For descriptions of the commands SET CMD OUT=, SET OUT=, and MEAS I488, refer to Appendix D.

Format

```
CLEAR I488;
TEST I488 MAXTIME=t line-parameter [error-statements;];
READ ID=I488(n) LENGTH=m 'format-characters' parameter-list ;
WRITE ID=I488(n) 'format-string' parameter-list [error-statements;];
SET I488 bus-commands [error-statements;];
SET CMD OUT='command-string' [error-statements;];
SET OUT='message-string' [error-statements;];
MEAS I488 INTO meas-variable;
```

where:	CLEAR/TEST/ READ/WRITE/ MEAS/SET	Command keywords. Must appear first in the statement.
	CLEAR	Asserts the IFC management line to reset all IEEE-488 instruments. The responses of specific instruments depend on how they implement the IEEE-488 reset function.

TEST	Tests the condition of line SRQ or EOI until either the line is asserted or the time-out (MAXTIME) expires. If by the end of the time-out the line has not been asserted, the error-statements execute. Defaults: SRQ, MAXTIME=32U
READ	First issues UNT, then MTA=n to quiet other instruments and to place the addressed instrument in talk mode. It then inputs characters from the instrument to the variables specified in the parameter-list, making any format conversions specified by the format-characters. Characters are accepted until either EOI is sent by the receiving device, LENGTH=m characters are received, a total of 80 characters are received, or a 1-second time-out expires after the last character. At the end of the transaction READ issues UNT. No error branching is available; if a time-out expires, the READ simply terminates. To read more than 80 characters, use MEAS I488 INTO, as described in Appendix D.
WRITE	Issues UNT, UNL, MLA=n to disable other instruments and to place the addressed instrument in listen mode. It then sends the characters specified by 'format-string' and parameter-list to the addressed instrument. If any character is not acknowledged within a 1-second time-out, a run-time error occurs. WRITE sends UNL at the end of the message. To send more than 80 characters, use SET I488 OUT=, as described in Appendix D.
I488	Keyword for the IEEE-488 Instrumentation Bus in CLEAR, TEST, MEAS, SET.
ID=I488(n)	Pre-defined channel identification for READ and WRITE. Addresses 1 to 30 are available for the external instrument. The instrument specified by n is addressed and placed in either listener (READ) or talker (WRITE) mode.
MAXTIME=t	Defines the time-out interval t in seconds for TEST. The value of t can be between 32U (microseconds) and 2 (seconds). Default: 32U
line-parameter	Specifies which I488 management line must be asserted for TEST to pass (i.e., not to execute the error-statements). EOI or SRQ can be specified. Default: SRQ

[error-statements;] The program performs the statements within the brackets when an error occurs. A TEST statement fails if the specified line (SRQ or EOI) is not activated within the time-out defined by MAXTIME=m. A WRITE statement fails if more than 1 second elapses before a character (or command) is sent.

LENGTH=m Gives the maximum number of characters that will be read from the instrument. After the mth character the command completes without waiting for EOI or another terminator. If LENGTH=m is not present, READ continues until the EOI line is activated or 80 characters are received.

'format-characters' A character string that specifies the order, types, and number of variables into which data from the addressed instrument are to be read. The usual variable types, e.g., %S%, are allowed. When reading into a %R% type of string, the READ command automatically appends a line-feed character when carriage-return is received. Refer to the READ command in para 6.2 for further information about how READ formats and type conversions function.

parameter-list A list of variable names in which each variable corresponds to a format character in the preceding 'format-characters' or 'format-string'. Refer to the READ and WRITE commands in para 6.2.

'format-string' A string of literal characters (e.g., 'V2.260E-2'), octal character values (e.g., %033%), or place-holders for variables (e.g., '%S%%S%F%'), mixed in any order, according to the programming requirements of the addressed instrument. The format-string typically specifies a type of measurement and range or a type of signal. Refer to the instrument's programming manual for the strings that should be sent. Refer to the command WRITE in para 6.2 for further information on how WRITE formats and type conversions function.

bus-commands A list of standard IEEE-488 mnemonics to be issued as bus commands. The mnemonics DCL, GET, LLO, MLA(n), MTA(n), MSA(n), SDC, SPD, SPE, UNT, and UNL are recognized; n specifies an instrument address from 1 to 30. Extended addresses must be implemented using the SET CMD OUT= syntax, described in Appendix D. Table 6-2 lists the standard mnemonics and their functions; however, it is advisable also to check the instrument's programming manual to find out if and how it responds to each of the standard commands.

CMD OUT='command-string'
OUT='message-string' Refer to Appendix D for descriptions of SET OUT= and SET CMD OUT= syntax. These statements are available for compatibility with earlier 2227X software releases and for unusual programming needs.

MEAS I488 INTO meas-variable

A form of the MEAS command; it uses a single variable of type string. Refer to Appendix D for further information

Table 6-2
IEEE-488 STANDARD BUS COMMANDS

Mnemonic	Description
DCL	Device Clear. Responses to this command may be implemented on the interface bus. Full implementation would cause the device to be reset or initialized and to be in a state only to monitor the bus. In effect, this implementation is a combination of the CLEAR I488 plus a special clear character in the device message-string. Other implementation would be a subset of the full implementation. Check the instrument's manual for IEEE 488 implementation specifications.
GET	Group Execute Trigger. Synchronizes function execution of all devices that have been set up. Some devices wait for this command; others do not.
GTL	Go to Local. Puts all devices that are on the bus in local or front-panel control rather than remote program control.
MLA(n)	My Listen Address. Where n is a number from decimal 0-30, places the corresponding I488 address (range octal 40-76) of the device that the controller wants to talk to on the bus. Address 0 is reserved for the controller listen address.
MTA(n)	My Talk Address. Where n is a number from decimal 0-30, places the corresponding I488 address (range octal 100-136) of the instrument to listen to on the bus. Address 0 is reserved for the controller talk address.
UNL	Unlisten. Disables all device listen modes. Assures that no device is listening for a unique command.
UNT	Untalk. Disables all device talk modes. Assures that no device is trying to talk.
LLO	Local Lock-Out. Locks out local or front-panel control. Some devices can respond to this command; some cannot.
MSA(n)	My Secondary Address. There may be a device with more than one data output. The secondary octal address range is identical to the primary talk address range (0-31).
SDC	Selected Device Clear. Resets or initializes any device that is in listen mode.

Examples

1. Consider an instrument that sends, first, a 3-character code indicating the type of measurement it has made; second, a 2-character code giving the range; and finally the measured value in floating-point format, with an ASCII sign character immediately preceding the ASCII numerals. Each of these codes is separated by a space character, and the string terminates with a carriage-return, for example,

```
DBM HI +43.21<CR>
```

   The following statements will assign the codes to appropriate string and float variables, which can then be processed further by the test program.

```
DECLARE STRING(3) MEASKIND;
DECLARE STRING(2) MEASRANGE;
DECLARE FLOAT MEASVAL;
READ ID=I488(1) '%S%%S%%F%' MEASKIND, MEARANGE, MEASVAL;
```

2. If an instrument returns a string format that is different from the 227X formats, string operations may be required to reformat the data. For example, an instrument may insert spaces between the sign character and the numerals when returning a numerical value. In such a case the sign must either be discarded (if not needed) or processed with special instructions. Consider an instrument that returns the same information as shown in example 1, but inserts spaces between the sign and the numerical values, for example,

```
DBM HI + 043.21<CR>
```

   The following statements would be needed to obtain the proper value in MEASVAL.

```
DECLARE STRING(3) MEASKIND;
DECLARE STRING(1) SIGN;
DECLARE STRING(2) MEASRANGE;
DECLARE FLOAT MEASVAL;

READ ID=I488(1) '%S%%S%%S%%F%' MEASKIND, MEASRANGE, SIGN, MEASVAL;
IF SIGN = '-' THEN
    LET MEASVAL = -1 * MEASVAL;
```

3. This example contains a sample test program for using a Fluke[1] 1953 Counter to measure the frequency of a signal on the UUT and to print the results. Included is an explanation of the major functions performed by the program, followed by a listing of the test program statements.

   a. Initializing the Program. DECLARE statements are needed to define each variable used in the program. Note that the proper length should be DECLAREd for the character string variable used to store character data from the instrument. If the DECLAREd variable specifies fewer characters than are actually received, any further characters will be not be accepted from the instrument. If the variable is DECLAREd longer

[1]John Fluke Mfg. Co., Inc., Mountlake Terrace, WA.

than needed, the system will time-out (after 1 s) and whatever data have been received will be accepted as the complete character string.

b. Connecting the Instrument to the UUT. The functional output (or input) of each IEEE-488 device plugs into BNC connectors on the rear panel of the processor bay. As shown in Figure 6-3, these connectors form the inputs to the optional instrument multiplexer MUX(2). These inputs can then be connected to the appropriate MUX channels (A, B, C, or D) via a SET MUX statement and the MUX channels connected to the appropriate test pins via a SET SCAN statement. These 2 types of statements are included in the sample program for reference. The connections specified by these statements are not actually needed in this program, however, since subsequent statements merely place the instrument in self-test mode and no UUT measurements are made.

c. Communication between the Workstation and I488 Device. A series of CLEAR, WRITE, TEST and READ statements directs the interaction between the I488 devices and the I488 controller in the test system. For this typical example, the communication takes place as follows:

1. The CLEAR I488 statement activates the Interface Clear (IFC) line to all I488 instruments on the bus.

2. The WRITE statement initializes all devices by first issuing UNL and UNT commands. MLA(1) specifies instrument number 1 as the device to which the control message is to be directed. In this example, the Fluke counter is assumed to be wired or set up to respond at address number 1. The WRITE statement then transmits a control message to the addressed instrument (the Fluke counter), informing it of the type of measurement to be made. This message is in the form of a character string tailored to the particular device being addressed. The particular sequence of characters needed to perform a desired function must be obtained from the instrument's programming or user manual.

In this example using the Fluke counter, the character string specifies the following functions:

F6	Self-test mode
R2	Range
A+1	Channel A setup information
LA+.5	Channel A voltage level information
B-1	Channel B setup information
LB+.5	Channel B voltage level information
S1M1H1	Operate mode
T	Trigger

3. The TEST I488 statement checks the I488 bus to determine if the Service Request (SRQ) line is active. MAXTIME specifies a time-out of 1 second; if the SRQ line is not activated within this time -- i.e., no service request apears -- the error branch executes, sending an error message to the console and aborting the test. If the service request appears within the time-out period, the next statement following TEST I488 is executed.

4. The READ statement changes the Fluke counter from listen to talk mode via an MTA(1) code and the 227X to listen mode by

MLA(0). The stream of characters from the counter is first read into the string variable Info, until either a space or tab is encountered or the 16th character is sent. Subsequent characters are treated as representing a float number, and are converted to binary floating-point form, to be stored in variable VAL.

The following is a listing of the sample test program along with a comment explaining the function of each statement.

```
DECLARE CSTRING INFO(16);                /* SAVE COUNTER OUTPUT STRING */
DECLARE VAL;                             /* STORE COUNTER DATA */
DECLARE BSTRING FAIL(1);                 /* SAVE TEST FAIL INDICATOR */
SET MUX AT (CHA=13:CHB=14:REF=13,14,23); /* SET INSTRUMENT MUX TO
                                         CONNECT COUNTER TO CHA AND CHB */
SET SCAN AT (CHA=8:CHB=5);               /* SET SCAN CHANNELS TO TEST PINS.
                                           (HERE FOR REFERENCE ONLY) */
CLEAR I488;                              /* SET THE IFC (INTERFACE CLEAR) LINE ON
                                         THE BUS. THE REACTION TO THIS
                                         COMMAND IS DEVICE DEPENDENT. */
WRITE ID=I488(1) 'F6R2A+1LA+.5B-1LB+.5S1M1H1T'
      [WRITE 'FAILED TO RECEIVE DATA FROM I488(1) %NL%';];
                                         /* ISSUE COMMAND STRING (UNIQUE */
                                         /* TO THIS INSTRUMENT) */
                                         /* FOR A COUNT MEASUREMENT */
TEST I488 MAXTIME=1 [                    /* CHECK BUS FOR SRQ. */
     WRITE '%NL% NO SRQ %NL%;            /* IF NOT SET, REPORT ERROR */
     BRANCH DONE;                        /* AND EXIT */
     ]
READ ID=I488(1) '%S% %F%' INFO, VAL      /* SENDS MTA=1, MLA=0 TO MAKE 227X */
                                         /* A LISTENER, INSTRUMENT A TALKER */
                                         /* STORE CHAR STRING IN INFO; */
                                         /* STORE COUNT */
                                         /* IN BINARY FLOAT FORM IN VAL */
WRITE 'TEST 1 FREQ=%G%' VAL;             /* DISPLAY THE RESULTS */

DONE:  END;
```

Rules and Considerations

1. The system software associated with the I488 bus option provides a 1-second time-out. Any time the system waits more than 1 s for a command or a response to be issued and completed, a time-out will occur. If the time-out occurs when the controller is in talk mode, a system run-time error results. If the time-out occurs when the controller is in listen mode, the incomplete response is accepted as being complete and the program continues. The most common occurrence of the time-out condition results when a character string is DECLAREd as n characters long, but fewer than n characters are actually placed on the bus by a device. The system will wait for n characters, until time-out occurs. One reason for this condition is that, when a device drops leading zeros from its measurement data, it sends fewer characters than DECLAREd.

2. When converting the received character string to a floating point number (via READ or WRITE), there must be no spaces between the sign character and the digits (e.g., + 1234.56 is wrong). If spaces do exist, the programmer must eliminate them before the conversion. This can be done by reading the character string into a string variable rather than a float, as shown in example 2 above. Alternatively, read the numerical response into a string variable. Then, using SUBSTR, extract the sign from the character string and then form a substring consisting of just the number. If necessary, the sign can be reinserted at the proper place. (Refer to the SUBSTR function in para 6.2 for forming the substring.)

3. The WRITE statement can send no more than 80 characters. If you need to send more than 80, use SET I488 (Refer to Appendix D).

4. The READ statement accepts no more than 80 characters. If a message will be longer than this, use MEAS I488 (Refer to Appendix D).

5. An I488 measurement, by itself, does not constitute a test. Additional statements will be needed to pass or fail the measurement, of the form

TEST ARITH VAL HI=nn LO=mm [<error-routine>;];

--------LGC (Digital Logic)---------

Refer to para 6.4 on Digital Testing.

--------MUX (Instrument Multiplexer)-------

General

Figure 6-2 illustrates the nature of the standard instrument multiplexer (MUX). System instruments are hardwired to the standard multiplexer inputs so that any of them can be switched to the 4 multiplexer output channels A, B, C, and D through a relay matrix. Note that the "low" side of each instrument and analog ground (REF 11) can be connected through relays to the "REFerence" bus. This bus can be connected to any one of the 4 MUX output channels as necessary, through REF 10. Analog ground (REF 11) is normally connected for in-circuit testing. MUX input REF 12 is not currently used. Multiplexer output channels A to D are switched through the pin scanner relays to the appropriate UUT pins.

As shown in Figures 6-3 and 6-4, the system may have up to 2 additional MUX boards, each with 12 input pins and 4 common output channels plus a reference. The standard MUX is connected to the standard source and measure units (DCS, DCM, ACZ). These MUX inputs are specified as pins 1 to 12 in test language statements. The other 2 MUX units are optional. The optional MUX defined by input pins 13 to 24 (Figure 6-3) has the 9 BNC connectors used for connecting external IEEE bus instruments to the UUT pins. The other optional MUX defined by input pins 25 to 36 (Figure 6-4) has optional source and measure units connected to its inputs.

Two test language statements control the MUX: SET MUX and CLEAR MUX. The SET MUX statement connects the specified instrument to the desired MUX A to D output channels. The output channels are referred to as CHA, CHB, CHC, and CHD. The reference bus is referred to as REF and does not connect directly to the pin scanner but, rather, connects the "low" side of each instrument to analog ground, as required. REF may be connected to one of the 4 output channels, however, by specifying MUX Input 10 (CHn=10, where n is one of the channels). MUX Input 10 has no external connections; it assigns the REF bus to the desired channel, as shown in Figure 6-2. A CLEAR MUX statement opens all the MUX relays.

Format:

SET MUX AT (pin-spec);

and

CLEAR MUX;

where: SET/CLEAR — Command keywords.

MUX — Keyword for Instrument Multiplexer(s). The keyword MUX is used to designate both the standard and optional multiplexers. Only the input pin specification determines which physical MUX board is being addressed.

AT (pin-spec) — A special parameter that specifies instrument connections to MUX output channels CHA, CHB, CHC, CHD, and REF bus. These connections may be stated explicitly or implicitly, as follows:

Explicit Connections:

(CHA=A-pins: CHB=B-pins: CHC=C-pins: CHD=D-pins:
REF=ref-pins)

Each of the 5 fields must be separated by a colon (:). To the left of each equal sign (=) is the MUX channel or reference designation; to the right of the = sign are the input pins assigned to that channel. Multiple pins are separated by commas. For example: SET MUX AT (CHA=4: CHB=5: CHC=10: REF=4,11).

Implicit Connections:

(A-pins: B-pins: C-pins: D-pins: ref-pins)

The channel and reference designations CHA, CHB, etc are not specified, but fields still appear in the same sequence (A, B, C, D, REF) and are still separated by a colon. Pins are assigned based on their position in a field. For example: SET MUX AT (4:5:10:REF=4,11)

Note that both explicit and implicit connections may be mixed in the same statement.

Examples

1. SET MUX AT (CHA=4: CHB=5: CHC=10: REF=4,11);

   Meaning: Connect the dc voltmeter inputs P+ (Input 4) and P- (Input 5) to MUX channels CHA and CHB, respectively (see Figure 6-2). Additionally, assign P- REFerence relay 4, along with Analog Ground (on REF 11), to connect these inputs to the REFerence bus. MUX input 10 (not connected to anything externally) is the REFerence bus and is connected to CHC. Thus P- and Analog Ground are connected to channel C.

2. SET MUX AT (CHA=8: CHB=9: CHD=10: REF=8,11);

   Meaning: Connect the dc ammeter (DCI DCM) with its MUX inputs 8 and 9 to MUX channels A and B. Channel D is connected to MUX Input 10 (the REF bus). REFerence relay 8 and analog ground 11 are connected to the REF bus and thus available to the pin scanner via CHD.

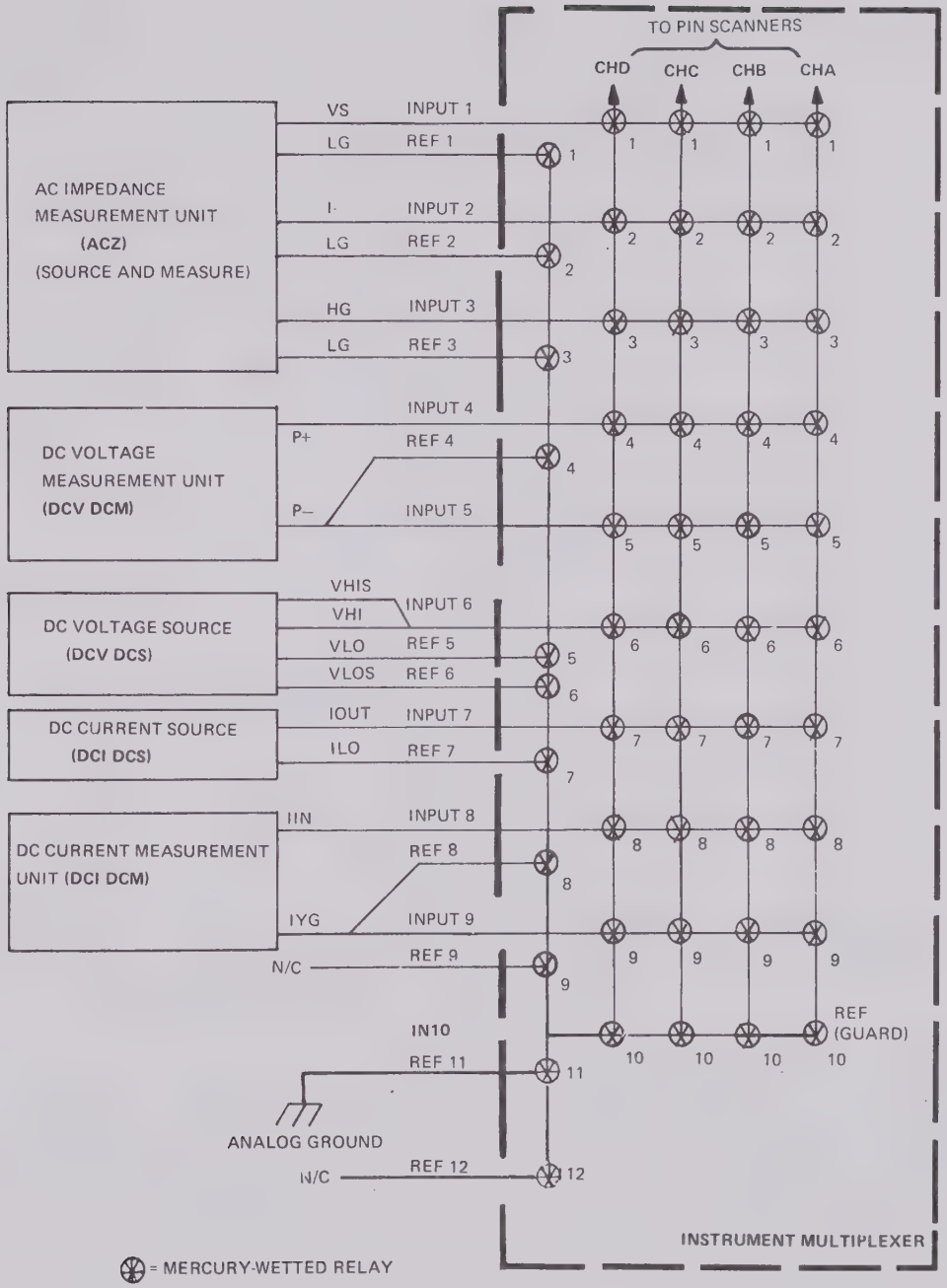


Figure 6-2. Standard Instrument Multiplexer MUX(1) Connections (Pins 1-12).

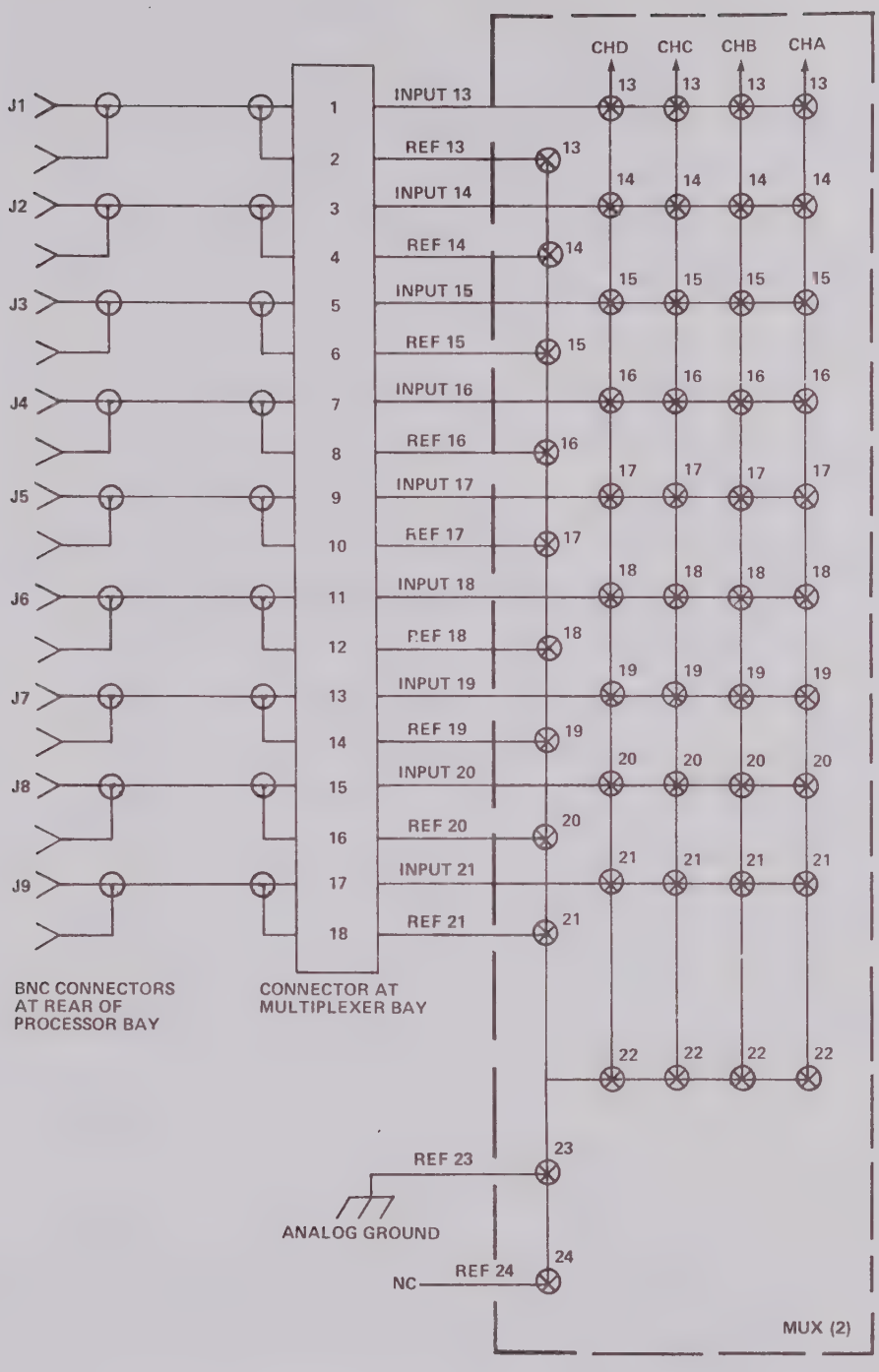


Figure 6-3. Optional Instrument Multiplexer MUX(2) Connections (Pins 13-24).

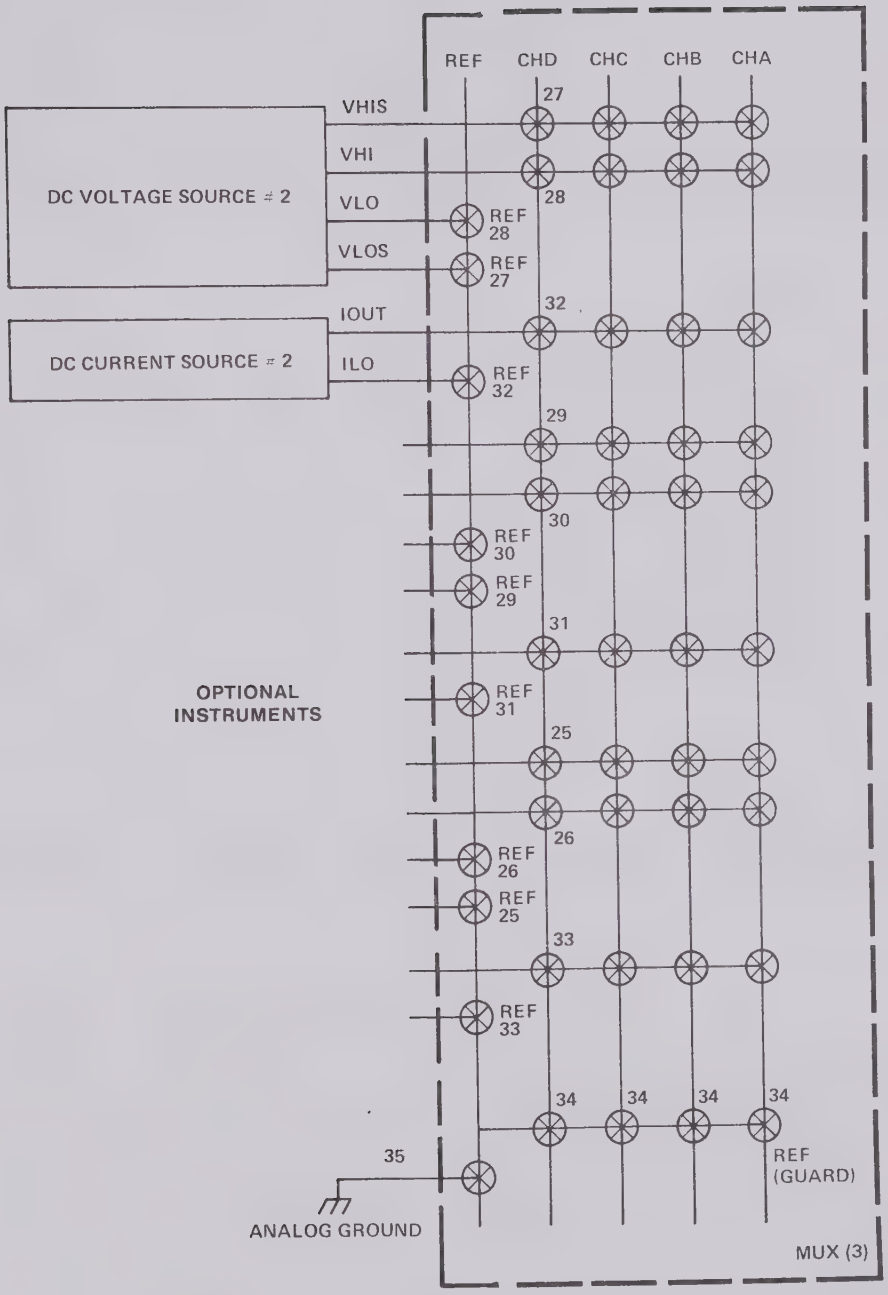


Figure 6-4. Optional Instrument Multiplexer MUX(3) Connections (Pins 25-36).

3. SET MUX AT (8: 9: CHD=10: 8,10);

   Meaning: This statement is partly implicit and partly explicit for the ammeter just described. Note that the CHn= for channels A, B and REF are not explicitly stated; they are implied by position. The specification to the left of the first colon lists the MUX inputs to be connected to CHA. The next specification (input 9 in this example) is connected to CHB. The succeeding specifications are for CHD and REF. Since there is no CHC specification, CHD has to be stated explicitly. Also, since the field following the CHD specification is REF, an implicit specification can be made.

Rules and Considerations

CAUTION

Between SET MUX statements, it is recommended that source instruments first be programmed to zero to discharge any reactive components connected through the pin scanner which might otherwise damage a multiplexer relay.

1. When a SET MUX statement is made, the connections are absolute and in effect until another SET MUX statement is made. Normally, an instrument is configured to the multiplexer output channels and then several pin scanner AT constructs or statements are executed for multiple tests or measurements.

2. All MUX connections are made in a break-before-make mode, i.e., all unspecified connections are opened before all specified connections are made.

3. A system can contain up to 3 MUX boards, as shown in Figures 6-2 to 6-4, with input pins referred to as 1-12 for the standard unit, 13-24 and 25-36 for the optional units. The same A, B, C and D output channels and REF terminal are common to all 3 units.

----------OPENS----------

General

The OPENS driver (TEST OPENS statement) is very similar to the SHORTS driver described later in this section. Both drivers test for shorts between pairs of nails in a designated list. The TEST OPENS statement, however, specifies groups of nails that should be shorted together (e.g., nails connected to a voltage track or ground track on the pc board) and reports a failure if no short is detected.

The method of testing shorts outlined in the SHORTS description also applies to the opens test, with the following exception: For the 2270, pure digital nails are all grouped together in the nail list (i.e., there is no distinction between "slow" digital and "fast" digital nails). However, a MAXDLY can still be specified, if the user wishes to delay the digital opens test for any reason.

The OPENS report, displayed when no continuity is detected between nails that should be shorted together, appears in the following format:

```
** OPEN PIN GROUPS **

(1,2) (4,5) (6,10)
      ^     ^
      |     |
      ----------------------NO continuity
```

This sample report indicates that nails 1 and 2 are shorted together, nails 4 and 5 are shorted together, and 6, 7, 8, 9, 10 are shorted together, but that there is no short between the nail groups shown in parentheses. If an IDD file exists, the report will also include a list of digital IC connections to the open nails.

Format

TEST OPENS AT (nail-list) parameters [error-command;];

where: TEST — Command keyword.

OPENS — Keyword for OPENS driver.

AT (nail-list) — Defines the nail set being tested for opens. All nails are considered hybrid for opens testing on a 2271 or 2272. Uses the following format:

(analog-nails:digital-nails)
or
(ANA=analog-nails:DIG=digital-nails)

ANA and DIG are optional keywords. As with other nail lists, nail numbers are separated by commas, and consecutive numbers can be expressed as a range of nails. For example:

(3,7,11:14-17) (ANA=1-4,55:DIG=20,22)

parameters — Allowable parameters are as follows:

I	Current range limit of the DCS (20 or 128 mA, default is 128 mA). Not applicable to pure digital nails.
R	Resistance threshold (5 - 1000 ohms, default is 10 ohms). Not applicable to pure digital nails.
VMAX	Voltage setting (up to 16 V, default is 0.1 V). Not applicable to pure digital nails.
MAXDLY	Maximum time to sustain a digital opens test before a pair of nails is considered not open (default is 0.27 ms).

INTO variable

Optional parameter that will cause the results of the shorts test to be stored in the specified bit string variable. Each bit position in the bit string represents a test nail (i.e., bit 1 corresponds to nail 1, bit 189 corresponds to nail 189, etc. A "1" indicates a failure (no short), a "0" indicates a short. If the bit string variable has been DECLAREd too small, a warning will be printed, but the system will attempt to continue the test and ignore the missing bit positions.

NOPRINT Optional parameter that can be used to suppress any diagnostic reports resulting from this test.

[error-command;] Optional. Statement executed upon test failure.

Example

1. TEST OPENS AT (15,17,20-23:37-40) R=20, MAXDLY=50M;

   Meaning: Test analog nails 15, 17, 20, 21, 22 and 23 and digital nails 37, 38, 39 and 40 for opens. Use 20 ohms as the opens threshold and maintain the opens test 50 ms before declaring that no open exists between a pair of nails.

Rules and Considerations

1. Since the analog continuity test is inherently more accurate than the digital test, hybrid nails should be included in the analog nail-list for more satisfactory results.

2. The parameter FAIL( ) in a TEST OPENS statement will suppress the diagnostic error report. However, since this feature may not be supported in future releases, the keyword NOPRINT should always be used to suppress the report.

----------PIO (Parallel I/O)----------

General

The PIO (Parallel I/O) commands allow the user to control and monitor the various devices on the Parallel I/O hardware (Relay Driver boards). These devices include relay drivers, TTL drivers, TTL sensors, TTL latches and TTL inverters, and are available to the user for implementing various functions on the UUT fixture. Input and output lines from Parallel I/O unit (1) are available at connector block 1 of the receiver unit, as described in the System Operation manual. The receiver unit provides the electrical and mechanical interface between the UUT fixture and the test system. The following table lists all available PIO devices and relates test language keywords with device mnemonics shown on the pc board schematic (see Diagrams Manual). Figure 6-5 shows a simplified block diagram of PIO(1) and its interconnections to the receiver connector block 1.

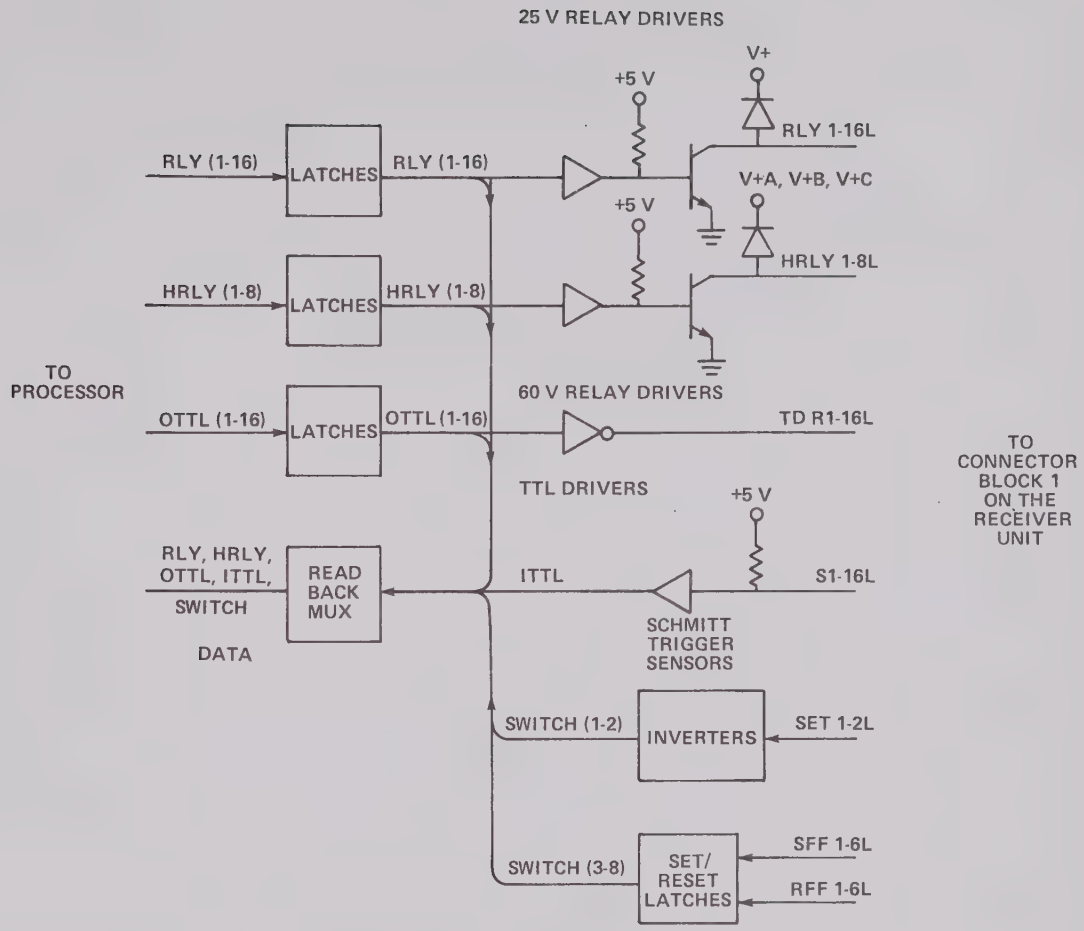


Figure 6-5. Parallel I/O Block Diagram.

Device	Mnemonic	Keyword (pin-list)
16 Relay Drivers (25V)	RLY1L-16L	RLY(1-16)
8 Relay Drivers (60V)	HRLY1L-8L	HRLY(1-8)
16 TTL Drivers	TDR1L-16L	OTTL(1-16)
16 TTL Sensors	S1-S16	ITTL (bits 1-16)
2 Inverters	SET1-2	SWITCH (bits 1-2)
6 Set/Reset Latches	SFF1-6	SWITCH (bits 3-8)

Note that keywords ITTL and SWITCH need no pin-list, since they specify read-only devices in which all pins are read back simultaneously. These PIO devices may be controlled and monitored by the following instrumentation commands:

SET PIO Activates the specified drivers or latches to the state indicated in the pin-specification. These states are latched and can be read back via a MEAS PIO statement. Only one set of devices (RLY, HRLY or OTTL) can be specified in a statement. Connection mode keywords CLOSE, OPEN, JAM, MBB, and BBM can be used to control the manner in which the actions are to be performed.

MEAS PIO Reads the latched states of all devices in the specified group and stores them in the indicated bit-string variable. Devices can only be read as a group and not as single devices (e.g., all 16 TTL sensors are read back at the same time). Individual positions in the bit-string can then be tested by using an IF-THEN-ELSE construct, the same as that described for FAIL bits in para 5.2. Example 5 illustrates the construct.

CLEAR PIO Resets most of the devices on the specified Parallel I/O unit to an inactive state. It does not, however, affect the state of the set/reset latches on the board.

Format

```
SET PIO (unit) RLY(pin-spec)/HRLY(pin-spec)/OTTL(pin-spec);

MEAS PIO (unit) measure-type INTO variable;

CLEAR PIO (unit);
```

where:

SET/MEAS/CLEAR	Command keywords. Must appear first in the statement.
PIO	Keyword for Parallel I/O (i.e., Relay Driver board).
(unit)	Optional. Unit (1), the default unit, refers to the optional board available for user functions and is the default device. Unit (0) is reserved for system use and should not be specified by the user during normal UUT testing.
RLY	Keyword for the 16 low-voltage (25-V) relays. Controls the state of the relay drivers in a SET statement and reads back their latched states in a MEAS statement.
HRLY	Keyword for the 8 high-voltage (60-V) relays. Controls the state of the relay drivers in a SET statement and reads back their latched states in a MEAS statement.
OTTL	Keyword for the 16 TTL driver outputs. Controls the state of the drivers in a SET statement and reads back their latched states in a MEAS statement.
ITTL	Keyword for the 16 TTL sensor inputs. Can be used only in a MEAS statement to read the sensor outputs.
SWITCH	Keyword for the 8 user latches on the board. Can be used only in a MEAS statement to read the state of the latches. Bits 1 and 2 contain the state of the set-only latches; bits 3-8 contain the state of the set/reset latches.
measure-type	Keyword for any one of the previously defined device types OTTL, RLY, HRLY, ITTL and SWITCH.
INTO variable	The state of the latches read by a MEAS PIO statement is stored in this bit-string variable.

pin-spec — Defines the state of each device in a SET PIO statement. The format is as follows:

(closing-condition # pin-list)

where: closing-condition

Optional keyword specifying the type of action caused by the SET PIO statement. May be any of the following:

NOTE

In the following description, "close" means to energize a relay, activate a TTL driver to a logic low output or set a latch to the logic one state, whichever is applicable to the device type. "Open" means to de-energize, deactivate or reset the device.

CLOSE - Close the indicated pins and leave the others of that type unaffected.

OPEN - Open the indicated pins and leave the others of that type unaffected.

JAM - Default condition. Close the indicated pins and open all others of that type, with no concern for the order in which they are opened or closed.

MBB - Make-before-break. First close the specified pins, then open all other pins of that type.

BBM - Break-before-make. First open all pins of the type not specified in the pin list, then close all pins specified in the list.

— Optional "all-but" construct. Complements the pin-list, causing the closing conditions to apply to all pins but those specified in the list.

pin-list

The list of pins to be acted upon by the closing-condition keyword. The complement of that list is used if the list is preceded by a #.

Examples

1. SET PIO RLY (CLOSE 1-3,5);

   Meaning: Close relays RLY1, 2, 3 and 5 on PIO unit (1) and leave relays RLY4 and RLY6-16 in their existing state.

2. SET PIO RLY (1-3,5);

   Meaning: Close relays RLY1, 2, 3 and 5 and simultaneously open relays RLY4 and RLY6-16.

3. SET PIO RLY (MBB 1-3,5);

   Meaning: Similar to Example 2 except that relays RLY1, 2, 3 and 5 are closed before relays RLY4 and RLY6-16 are opened.

4. SET PIO RLY (MBB # 1-3,5);

   Meaning: Close relays RLY4 and RLY6-16 first; then open relays RLY1-3 and RLY5.

5. DECLARE BSTRING TTLSENS(16);
   '
   '
   MEAS PIO ITTL INTO TTLSENS;
   IF TTLSENS(6) THEN BRANCH TS6;

   Meaning: Read the 16 TTL sensor inputs into bit-string variable TTLSENS. If bit 6 is a logic 1, showing that TTL sensor to be active (low), branch to the statement labeled TS6.

6. SET PIO RLY( );

   Meaning: Open all relays (1-16).

7. SET PIO RLY(#);

   Meaning: Close all relays (1-16).

Rules and Considerations

1. CLEAR PIO does not reset the latches specified by the keyword SWITCH. The user can only initialize these latches to a known state externally, via the latch input lines.

2. A MEAS PIO statement reads back the state of all devices in the specified group. Single devices within the group (e.g., ITTL(5)) cannot be read back individually. However, the bit-string construct described in para 5.2 for analyzing the fail vector can also be used for testing individual bits in the MEAS PIO bit-string variable.

3. When reading the states of the TTL sensors via a MEAS PIO statement, logic ones in the bit string signify active low sensor inputs, and logic zeros in the bit string signify inactive high sensor inputs.

4. No more than one device type (RLY, HRLY or OTTL) and one connection keyword (e.g., CLOSE, OPEN, etc) can be used in a SET PIO statement.

5. The software waits approximately 3 ms after latching PIO data (either for input or output) before proceeding with the next statement. A 3-ms delay is also included between opening and closing pins in an MBB or BBM specification.

6. When relays are driven by a PIO command, there will be a finite amount of time (often in the 2ms to 20ms range) for the relay contacts to settle. A DELAY statement should be inserted after a PIO command to allow sufficient settling time before the next command.

----------RM (Ohmmeter)----------

General

Resistance can be tested or measured by means of a single TEST R or MEAS R statement. The ohmmeter uses the system dc voltage source and dc current meter and then obtains the resistance value by automatically dividing the applied voltage by the resulting current.

Since the dc voltage source (see DCS) is used as part of the ohmmeter circuit, the current limit for the source is in effect. It can be specfied using the I= parameter as either 0.02 A or 0.128 A. If not specified, I= defaults to 0.128 A. (The current limits are approximate.)

The voltage to be sourced can be specified with the V= parameter up to 16 V. If not specified, the voltage defaults to 0.2 V. It is important that voltage specification take into consideration the expected resistance value, so that current limit is not reached. For example, if 10 V were specified across 50 ohms, current limiting would occur, and the resistance measurement would be in error. Any measurement made in current limiting is reported as 0.0 ohm.

Format

TEST/MEAS R parameters [error-command;];

where:

TEST/MEAS	Command keywords for a TEST or MEAS statement must appear first.
R	Keyword indicating a resistance measurement.
parameters	Allowable parameters are AT ( ), V= , I= , DLY= , RDLY= , limits= , MAX= , INTO variable. and FAIL ( ). A MEAS R statement requires the INTO variable parameter to store the measured value, and a TEST R statement requires limits (HI= and LO=, or VAL= and TOL=). All other parameters are optional.
[error-command;]	Optional. May contain any test language statement (other than burst-only command). This error command is executed if the instrumentation statement fails.

Parameter Definitions

V=	The applied voltage specified in volts up to 16 V max. Default is 0.2 V.
I=	Current limit in amps. May be either 20 mA or 128 mA. Default value is 128 mA.
limits=	May be expressed as a high limit (HI=), low limit (LO=), or both, or may be specified as a nominal value (VAL=) with tolerance (TOL=).
DLY=	Set-up delay in seconds. Maximum value is 1.999 s; default value is 0. Specifies the minimum time allowed between setting up the source and making a measurement.

RDLY= Delay (in seconds) between each range change of the ohmmeter. Maximum value is 1.999 s; default value is 0.

MAX= Specifies a floating-point value that defines the starting point for auto-ranging.

INTO variable Measurement results are stored in the variable specified by this parameter.

FAIL(n) Optional. Fail vector bit n. Set to 1 if a failure is detected. If n is a null (i.e., FAIL( )), then no fail bit is set. If FAIL(n) is not specified at all, then FAIL(1) is assumed.

AT( ) This parameter is identical to the AT( ) construct in a scanner statement and performs the same function. It connects the MUX output channels and the reference bus to the appropriate UUT pins.

NOTE

Prior to executing a TEST R or MEAS R statement, proper MUX connections to the ohmmeter must be made via the following statement:

```
SET MUX AT(6:8:9:10:5,6,11);
```

This connects the dc voltage source (DCS) to CHA, the dc current measure unit (DCM) to CHB, the DCM guard sense to CHC, and the analog ground to CHD.

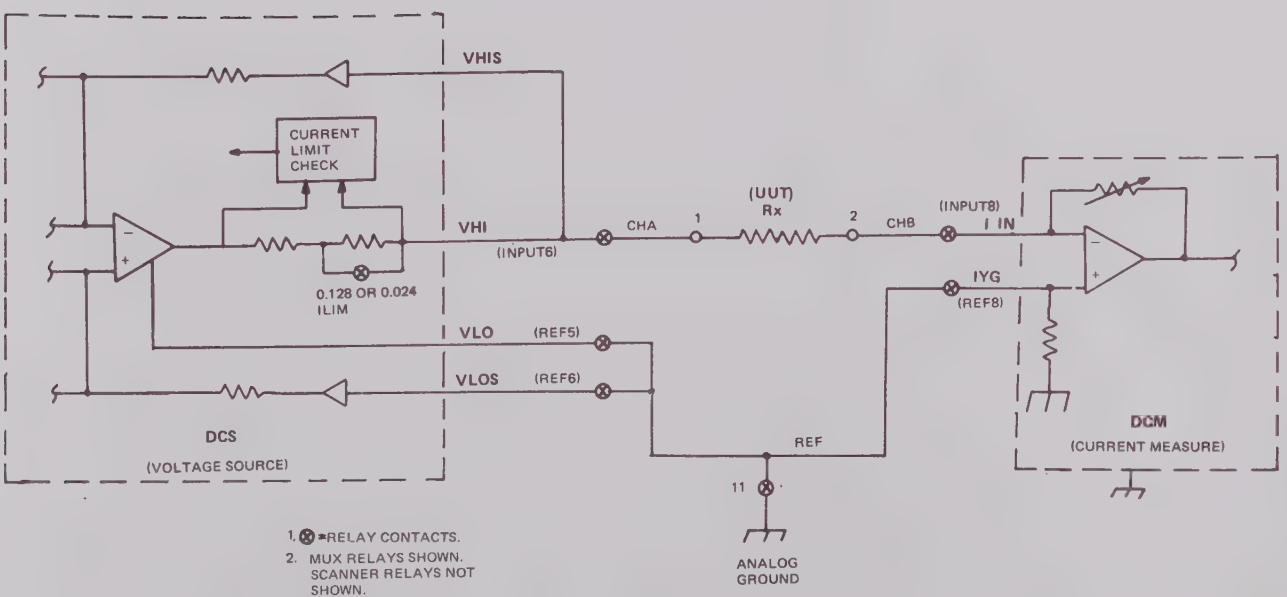


Figure 6-6. 2-Terminal Resistance Measurement (Series, Non-Guarded).

Examples

1. MEAS R INTO RVAL AT(1:2) [BRANCH RERR;];

   Meaning: Figure 6-6 illustrates this typical ohmmeter connection for making 2-terminal non-guarded measurements. The resistance being measured is connected across scanner pins 1 and 2. A known voltage (defaulted to V=0.2 in this example) is sourced across Rx by applying VHI to pin 1 and VLO to analog ground via the REFerence bus. Analog ground is reflected to scanner pin 2 as virtual ground through the current meter input amplifier.

2. MEAS R INTO RESVAL V=2.5 AT(1:2:3:3) [BRANCH RFAIL;];

   Meaning: Figure 6-7 illustrates a typical ohmmeter connection for making guarded resistance measurements. The resistance to be measured is connected across scanner pins 1 and 2. A known voltage (2.5V) is sourced across Rx by applying VHI to pin 1 and VLO to both analog ground and to the guard point (scanner pin 3) via the REFerence bus on CHD. Analog ground is reflected to scanner pin 2 as virtual ground through the current meter amplifier.

3. MEAS R INTO R1 DLY=50.M V=5.0 AT(4:5:6:6) [BRANCH H100;];

   Meaning: The resistance is measured between scanner pins 4 and 5 with pin 6 guarded. (The MUX is connected through default assignments.) The measured value is stored in variable R1. The stimulus is set to 5 V and, after a 50-ms delay, the current is measured. If the measurement cannot be made, the error branch is taken.

4. TEST R HI=5.5K LO=4.5K [BRANCH ERR1;];

   Meaning: Test the resistance between the scanner pins defined by the most recent SET SCAN statement, and check that it falls within the specified limits (4.5K to 5.5K).

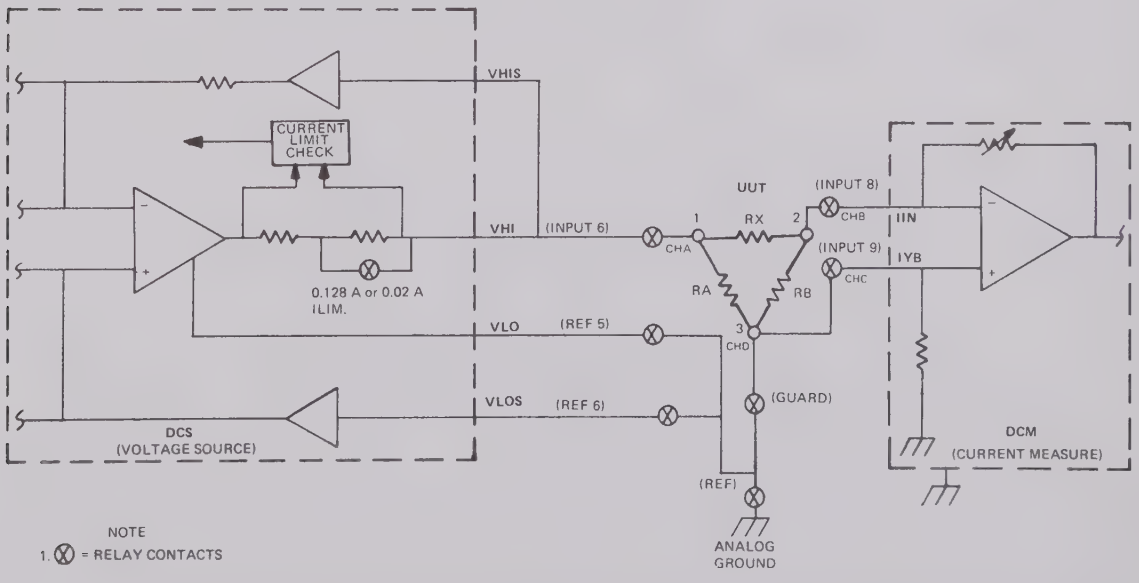


Figure 6-7. DC Resistance Measurements (Parallel, Guarded).

Rules and Considerations

1. When low resistance values are to be measured, it is suggested that the measurement be performed with separate source voltage (DCS) and measure (DCM) statements. This is to overcome the effects of the system multiplexer and scanner resistances that are more significant where lower-resistance measurements are concerned. With discrete statements, the voltage across the load can be checked and the voltage source adjusted until the voltage across the load reaches the appropriate value.

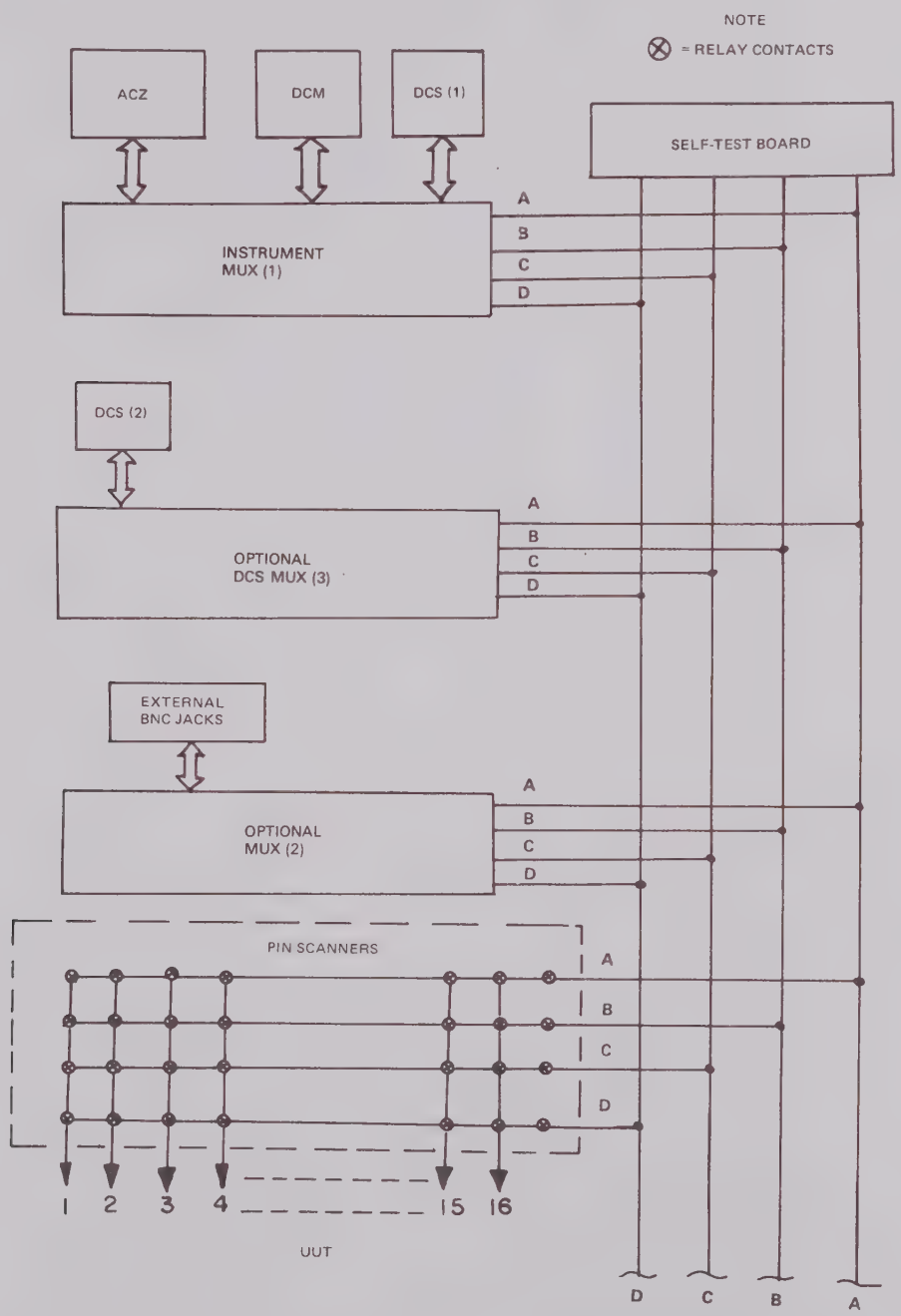


Figure 6-8. Multiplexer-To-Pin Scanner Connections.

----------SCAN (Pin Scanner)----------

General

The pin scanner shown in Figure 6-8 consists of a relay matrix which is used to connect the 4 output channels from the instrument multiplexer (MUX) to the UUT nodes. Each pin scanner board on the 2270 and 2272, or scanner-driver/sensor board on the 2271 and 2275, can connect MUX channels A, B, C or D to any of 16 test nails.

Any analog instrument may be applied to a UUT pin by connecting that instrument to a MUX output channel (A, B, C, or D) and then connecting that MUX channel to the test pin via the SCANner. The SCANner is used only for analog (and shorts) testing. Digital testing is accomplished through the Driver/Sensor system.

Two types of instrumentation statements control the pin scanner: SET SCAN and CLEAR SCAN. A SET SCAN statement connects the specified MUX outputs to the specified UUT pins. The CLEAR SCAN statement opens all scanner relays.

Format

SET SCAN AT(pin-list);

and

CLEAR SCAN;

where: SET/CLEAR — Command keywords.

SCAN — Keyword for pin scanner.

AT(pin-list) — A special parameter that specifies SCANner input connections to the UUT pins. The format is similar to the pin-connection spec used for MUX statements. Four fields are defined (CHA, CHB, CHC, and CHD), each separated by a colon (:). Pin assignments may be made explicitly (e.g., CHA=22: CHB=25:, etc) or implicitly (e.g., 22:25:, etc).

A closing-condition keyword may also be included in the pin-list to indicate the closing and opening operations of the scanner relays. The appearance of one of these keywords in the pin-list affects all pins in the list. As shown in Example 2, the keyword should appear just before the pin numbers.

The function of each keyword is as follows:

CLOSE - Close the indicated pins and leave the others unaffected. Not available on the 2272.

OPEN - Open the indicated pins and leave the others unaffected. Not available on the 2272.

JAM - Default condition. Close the indicated pins and open all others, with no concern for the order in which they are opened or closed.

MBB - Make-before-break. First close the specified pins, then open all others. Not available on the 2272.

BBM - Break-before make. First open all pins not specified in the pin list, then close all pins in the list. Not available on the 2272.

There is also an optional "all-but" construct (not available on the 2272) specified by a # symbol. A # before the pin list complements the list, causing the closing-conditions to apply to all pins but those specified in the list.

A scanner pin-list may also be defined by a bit string variable, (not available on the 2272) in which a 0 specifies an open pin and a 1 specifies a closed pin. Each bit in the bit string corresponds to a scanner pin in the pin list. The leftmost bit (bit 1) controls pin 1, the next bit controls pin 2, etc. For additional examples of scanner variables, refer to the description of the BITSET and BITCLR statements.

Examples

1. SET SCAN AT(CHA=5-8:CHB=6,9,10);

   Meaning: Connect scanner pins 5 through 8 to MUX Channel A and pins 6, 9, and 10 to MUX Channel B.

2. SET SCAN AT(CHA=MBB 17,18,20:CHB=16,19);

   Meaning: First connect pins 17, 18 and 20 to CHA; and connect pins 16 and 19 to channel B. Then disconnect all other pins.

3. SET SCAN AT(1:3-6:9:12);

   Meaning: Connect scanner pin 1 to CHA, pins 3 through 6 to CHB, pin 9 to CHC and pin 12 to CHD.

4. SET SCAN AT(1,2:CHC=3-6);

   Meaning: This statement is part implicit and part explicit. Connect pins 1 and 2 to CHA, pins 3-6 to CHC.

5. SET SCAN AT(1,2::3-6);

   Meaning: Same as Example 4. A null in the CHB field indicates that no pins are connected to channel B.

6. SET SCAN AT(CLOSE #);

   Meaning: Close all scanner pins.

7. SET SCAN (BSA:BSB);

(Note: This examples applies only to a 2270 system.)

Meaning: BSA and BSB must be 2 previously defined bit-string variables. All scanner pins corresponding to 1's in BSA are connected to MUX channel A; all pins corresponding to 1's in BSB are connected to MUX channel B.

8. CLEAR SCAN;

Meaning: Open all scanner relays, disconnecting all UUT pins from the MUX output channels.

Rules and Considerations

1. The scanner connection-mode keywords (MBB, BBM, OPEN, etc) should be used whenever reactive load conditions exist to avoid current spikes that could damage scanner relays. For example, to prevent a current spike from being developed when a scanner pin is opened to a reactive load, either MBB could be used or the reactive load could be discharged with a source before any pin scanner changes are made.

CAUTION
Between SET SCAN statements it is recommended that any source instruments first be programmed to zero to discharge any reactive components connected through the pin scanner, which could otherwise possibly damage a scanner relay.

2. Pin scanner AT( ) constructs can be included in both TEST and MEAS statements for the ACZ and RM instruments as follows:

```
MEAS CP ACZ INTO C1 AT(1 : 2 : 3 : 3);
```

3. For dual fixture applications, scanner pins assigned to one fixture should be offset by a multiple of 16 from those pins assigned to the other fixture. (See NAILOFFSET description earlier in this section.)

--------SHORTS (Shorts Detector)--------

General

The SHORTS driver is a software test module that uses the system hardware to check for unwanted shorts between nails listed in a TEST SHORTS statement. To prevent shorts from masking out or being interpreted as component failures, continuity checks (SHORTS and OPENS tests) should always be run before any other component test in a test program.

Shorts testing is accomplished on the 2271, 2272, and on 2270 analog and hybrid nails through the use of the voltage source and current measurement units along with the MUX and nail scanners. Shorts testing on 2270 digital nails is performed via current sources at the output of each driver/sensor. If a short is detected between 2 nodes, the system verifies that the indication does not result from a momentary short because of a reactive component or from a fuse or jumper connection on the board.

As shown in the Format description, a TEST SHORTS statement has a nail-list containing the nails to be shorts tested. In the 2270, this nail list must be divided into 3 sections: All analog (or hybrid) nails are listed together, followed by all "fast" digital nails, then all "slow" digital nails. The actual shorts testing takes place as follows:

1. Test All Hybrid (or Analog) Nails. Nail pairs in this list are tested for shorts by applying a dc voltage (VMAX) from the DC Source (DCS) to one nail and testing the resulting current with the DC Measure (DCM) at the other nail. If the current is greater than VMAX/R, and the system verifies that the shorts indication is not caused by a reactive component (e.g., a charging capacitor) or by a fuse or jumper, then a shorted nail is assumed. The default shorts threshold (R) is 10 ohms and the default VMAX is 100 mV, resulting in a threshold current of 100 mV/10 ohms = 10 mA.

NOTE

If other values of VMAX and R are specified, those values should be chosen so that the threshold current (VMAX/R) will not exceed 40 mA.

2. Test All Digital Nails (2270 only). All nail pairs in the digital nail list (both "fast" and "slow" nails) are tested for shorts via current sources at the output of each driver sensor. The user has no control over these current sources. They are activated internally whenever digital nails are specified in a TEST SHORTS statement.

The shorts resistance threshold for digital nails is approximately 40±30 ohms and cannot be changed by the user. Within this range, the threshold may vary between systems, but the threshold value within each system remains constant.

For all nails in the fast digital group (i.e., those digital nails not connected to large capacitors), a shorts/no-shorts decision is made in approximately 300 us. For nails in the slow digital group, a longer delay may be used, as specified by the MAXDLY parameter.

3. Test For Analog-to-Digital Shorts (2270 only). After each group of nails has been tested separately, another test is performed to check for shorts between analog and digital nails. A 200 mV source is applied to each analog nail and the sensors on the digital nails are checked for a "fake" 0 state. As in the case of the digital shorts test, the threshold resistance is approximately 40±30 ohms, and the delay is 300us (or MAXDLY for slow nails).

When shorts testing is complete, unless otherwise specified by the user, the system displays a shorts report in the following format:

```
** SHORTED NAILS **
1, 3, 7-9, 53;
2, 6;
4, 51;
```

Each line in the report (i.e., each list terminated by a semicolon) indicates a group of nails that were found shorted together. In this example, nails 1, 3, 7, 8, 9, 53 are shorted together; nails 2 and 6 are shorted together; and nails 4 and 51 are shorted together. If an IDD file exists, the report will also include all connections to the shorted nails.

To inhibit displaying the report, the parameter NOPRINT can be included in the TEST SHORTS statement.

Format

TEST SHORTS AT(nail-list) parameters [error-command;];

where: TEST — Command keyword. Must appear first in the statement.

SHORTS — Keyword used to specify continuity testing.

AT(nail-list) — Defines the nail set to be tested for shorts. The list may contain discrete (non-inclusive) nails, each separated by a comma, for example, AT(1,7,13,18,19), or it may specify a set of contiguous nails such as AT(20-29), or a combination of both. (See Examples).

In a 2270 system with both analog and digital nails, the nail list must distinguish between those nails checked by an analog shorts test and those checked by a digital shorts test. The following format applies:

```
AT(ANA=analog-nails: FDIG="fast"-digital-nails:
SDIG="slow"-dig-nails)MAXDLY=value;
```

or

```
AT(analog-nails: digital nails:
delayed-dig-nails)MAXDLY=value;
```

where: ANA - Keyword specifying analog shorts test.

FDIG - Keyword specifying "fast" digital shorts test.

SDIG - Keyword specifying "slow" digital shorts test (i.e., a test that may be affected by the presence of capacitive loading). Connectivity measurements for all nails in this list continue until continuity is detected or until MAXDLY time has elapsed.

If the keywords ANA, FDIG and SDIG are used, the order is unimportant. If the keywords are not used, the nail lists must appear in the order shown.

parameters — Allowable parameters for this type of statement are: AT(nail-list) [described above], I=, R=, VMAX=, INTO variable, NOPRINT and FAIL( ). With the exception of AT(nail-list), all other parameters are optional.

[error-command;] — Optional. May contain any test language statement (other than those used only in bursts). This error command is executed if the instrumentation statement fails and if the CONtinue test after shorts option has not been selected.

Parameter Definitions

R=	Shorts threshold resistance in ohms. Must be between 5 and 1000 ohms. Default value is 10 ohms. Not applicable to pure digital nails.
VMAX=	Applied dc voltage in volts, up to 16 V max. Default value is 0.1 V. Not applicable to pure digital nails.
I=	Current limit in amps of the DS voltage. May be either 20 mA or 128 mA. Default value is 128 mA. Not applicable to pure digital nails.
MAXDLY=	Specifies how long the system will wait before declaring a slow digital nail to be shorted. Maximum of 2 seconds.
INTO variable	Optional parameter that will cause the results of the shorts test to be stored in the specified bit string variable. Each bit position in the bit string represents a test nail (i.e., bit 1 corresponds to nail 1, bit 189 corresponds to nail 189, etc). A "1" indicates a failure (an unwanted short), a "0" indicates no short. If the bit string variable has been DECLAREd too small, a warning will be printed, but the system will attempt to continue the test and ignore the missing bit positions.
NOPRINT	Optional parameter that can be used to suppress any system supplied diagnostic reports resulting from this test.
FAIL(n)	Optional. Fail vector bit n. Set to 1 if a failure is detected. If n is a null (i.e., FAIL( )), then no fail bit is set. If FAIL(n) is not specified at all, then FAIL(1) is assumed.

Examples

1. TEST SHORTS R=75 AT(1-16) FAIL(3);

   Meaning: Perform an analog shorts test between all nail pairs from nail 1 to nail 16. Record a short between all nail pairs with a resistance of less than 75 ohms. Default voltage is 0.1 V. If any shorts are detected, set FAIL bit 3.

2. TEST SHORTS R=100 AT(1-256) VMAX=1.0 NOPRINT;

   Meaning: Set the DCS to 1 V and perform an analog shorts test between all nail pairs between 1 and 256. If the resistance is less than 100 ohms between any set of nails, set FAIL(1). Do not print a shorts report.

3. TEST SHORTS AT(1-9,13:27,30-51:59-62) MAXDLY=35M;

   Meaning: Perform an analog shorts test on nails 1-9 and 13, and perform a digital shorts test on nails 27, 30-51 and 59-62. However, don't consider an "open" on nails 59-62 as a shorts failure until after a 35-ms delay.

4. TEST SHORTS AT(FDIG=12-25);

   or

   TEST SHORTS AT(:12-25);

   Meaning: Both statements are equivalent and perform a digital shorts test on nails 12-25.

Rules and Considerations

1. To prevent current limiting of the DCS and still allow a sufficient current swing for charging capacitors, always keep VMAX/R < 40mA.

   NOTE
   If a test program implemented prior to Release 5 has a value of VMAX/R greater than 40 mA, you must re-adjust VMAX and/or R to a 40 mA limit before re-translating that program on new (Release 5 and over) software.

2. Since the analog continuity test is inherently more accurate than the digital test, hybrid nails should be included in the analog nail-list for more accurate results.

3. If the CONtinue test after shorts option in DEBUG or DIAGNOSE mode is selected, the [error-command;] in a TEST SHORTS statement will be ignored.

----------STM (Self-Test Module)----------

General

The STM (Self-Test Module) commands allow the user to control and monitor the various devices on the Self-Test Module. This board contains 42 relay drivers (and relays), 16 light-emitting diodes (LEDs) and 12 manually operated sense switches. The relays can be used to connect system sources and measurement units to various components on the board, and thus verify measurement capabilities of the system. The components are mounted in such a manner that various types of guarded and unguarded measurements can be made by specifying the proper relay connections. The board has access to 4 analog channels (A, B, C and D) via 4 normally open relays, and the 4 channels can be connected to external terminals via 4 other relays. The 16 LEDs are connected to the latched bus outputs and therefore allow the user to verify the operation of the system bus. The sense switches permit user interaction with the program.

Refer to Figure 6-9 for detailed connection information on each STM device. The following table relates test language keywords to device mnemonics (signal name) found on the schematic diagrams in the Diagrams Manual.

Relays and Relay Drivers (42)

KEYWORD	MNEMONIC	KEYWORD	MNEMONIC	KEYWORD	MNEMONIC
RLY(1)	K1A	RLY(17)	K1B	RLY(33)	K1C
RLY(2)	K2A	RLY(18)	K2B	RLY(34)	K2C
RLY(3)	K3A	RLY(19)	K3B	RLY(35)	K3C
RLY(4)	K4A	RLY(20)	K4B	RLY(36)	K4C
RLY(5)	K5A	RLY(21)	K5B	RLY(37)	K5C
RLY(6)	K6A	RLY(22)	K6B	RLY(38)	Not used
RLY(7)	K7A	RLY(23)	K7B	RLY(39)	K7C
RLY(8)	K8A	RLY(24)	K8B	RLY(40)	K8C
RLY(9)	K9A	RLY(25)	Not used	RLY(41)	K9C
RLY(10)	K10A	RLY(26)	K10B	RLY(42)	K10C
RLY(11)	K11A	RLY(27)	K11B	RLY(43)	K11C
RLY(12)	K12A	RLY(28)	Not used	RLY(44)	K12C
RLY(13)	K13A	RLY(29)	Not used	RLY(45)	K13C
RLY(14)	K14A	RLY(30)	K14B	RLY(46)	K14C
RLY(15)	K15A	RLY(31)	K15B	RLY(47)	K15C
RLY(16)	Not used	RLY(32)	Not used	RLY(48)	K16C

LEDs (16)

KEYWORD	MNEMONIC	KEYWORD	MNEMONIC	KEYWORD	MNEMONIC
LED(1)	CR3	LED(6)	CR8	LED(11)	CR13
LED(2)	CR4	LED(7)	CR9	LED(12)	CR14
LED(3)	CR5	LED(8)	CR10	LED(13)	CR15
LED(4)	CR6	LED(9)	CR11	LED(14)	CR16
LED(5)	CR7	LED(10)	CR12	LED(15)	CR17
				LED(16)	CR18

Sense Switches (12)

KEYWORD	MNEMONIC
SWITCH	(Bits 1-12 correspond to the switches that are tied to bus lines DIN00-11 on the STM.)

Note that, since the SWITCHes are read-only devices, a pin-list is not needed. The state of all 12 switches is read back simultaneously. These STM devices may be controlled and monitored by the following instrumentation commands.

SET STM Activates the specified device as indicated by the pin-specification. The specified states are stored in latches on the STM and the latch outputs drive the relays or LEDs. Connection modes CLOSE, OPEN, JAM, MBB and BBM may be specified in the same manner as in the PIO statements. No more than one device may be specified in the same SET STM statement.

MEAS STM Reads back the latched state of the relays LEDs or sense switches into a bit-string variable, as specified in the statement.

CLEAR STM Resets all latches on the STM, thereby de-energizing all relays and turning off all LEDs.

Format

```
SET STM RLY(pin-spec)/LED(pin-spec);

MEAS STM measure-type INTO variable;

CLEAR STM;
```

where: SET/MEAS/CLEAR Command keywords. Must appear first in the statement.

STM — Keyword for the Self-Test Module.

RLY — Keyword for the 48 relays on the STM. RLY(pin-spec) controls the individual relay drivers in a SET statement; RLY reads back their latched states in a MEAS statement.

LED — Keyword for the 16 light-emitting diodes on the STM. LED(pin-spec) controls the individual LEDs in a SET statement; LED reads back their latched states in a MEAS statement.

measure-type — May be any one of the following keywords: RLY, LED or SWITCH. RLY and LED are described above. SWITCH is a keyword for the 12 sense switches on the STM. These switches may only be read back in a MEAS statement.

INTO variable — The states of the specified devices in a MEAS statement are stored in this bit-string variable.

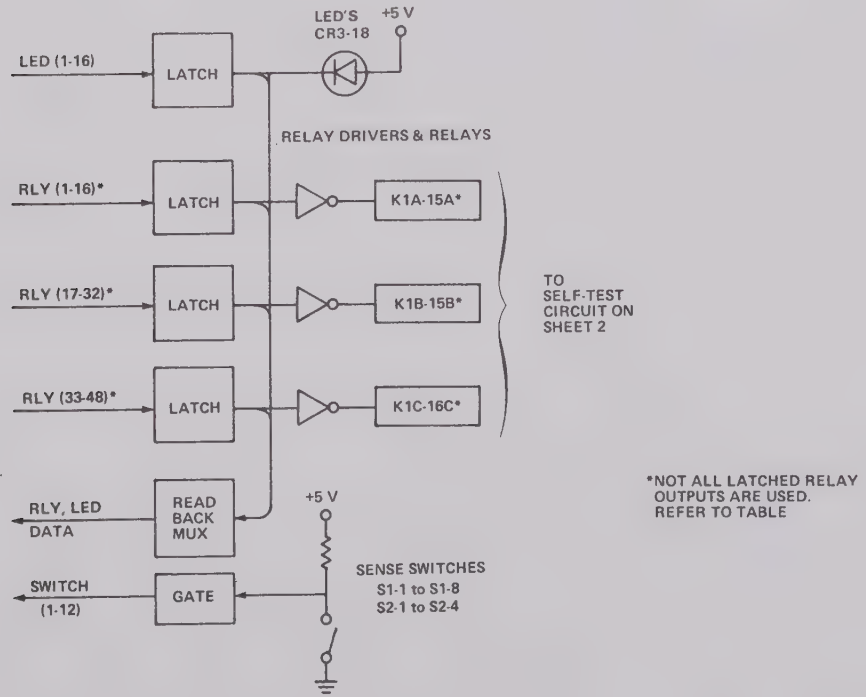


Figure 6-9. Self-Test Module (STM) Block Diagram (Sheet 1 of 2).

pin-spec Defines the action of the pins in a SET STM statement. The format is identical to that described in Parallel I/O (PIO) and consists of:

(closing-condition # pin-list).

NOTE

A SET STM RLY( ) will open all relays and a SET STM LED( ) will turn off all LEDs.

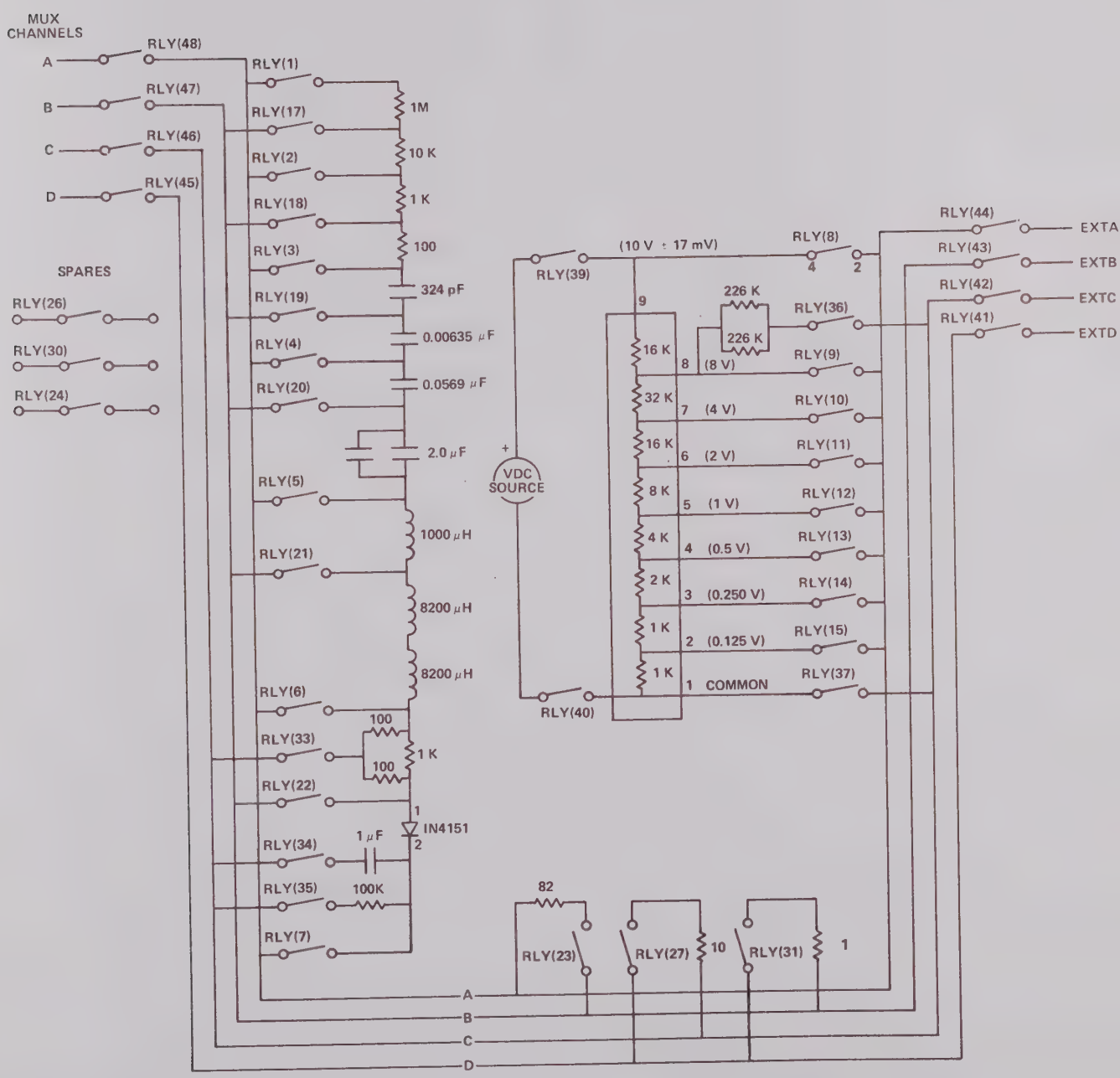


Figure 6-9. Self-Test Module (STM) Block Diagram (Sheet 2 of 2).

Examples

1. SET STM LED (1-16);

   Meaning: Turn on all 16 LEDs on the STM.

2. SET MUX AT(CHA=5:CHC=4);
   SET STM RLY (39,40,48,46,9,37);
   MEAS DCV INTO VVAL VAL=-8 TOL=1 [BRANCH FN1;];

   Meaning: This group of statements tests the DC Voltmeter (DCV) by applying a reference voltage from a source on the STM across a resistor network, also on the STM, and measuring the known voltage with the system DCV (see Figure 6-10). Figure 6-10 shows that portion of the STM circuit in Figure 6-9 that is involved in this example. Each of the 3 instructions performs the following functions:

   The SET MUX statement connects the DC Voltmeter to MUX channels A and C.

   The SET STM statement closes the relays indicated on Figure 6-10 and opens all others (not shown on the diagram). This applies an internal 10-V source across a resistor network and connects MUX channels A and C (containing the voltmeter) across the 8-V point in the network.

   The MEAS DCV statement stores the DCV measurement in VVAL and causes a branch to FN1 if the measured value falls outside the range of -8 V ±1%.

Rules and Considerations

1. A MEAS STM statement reads back the state of all devices in the specified group. Single devices within the group (e.g., RLY(4)) cannot be read back individually. However, the bit-string construct described in para 5.2 for analyzing the fail vector can also be used to test individual bits in the MEAS STM bit-string variable.

2. No more than one device type (RLY or LED) and one connection keyword (e.g., CLOSE, OPEN, etc) can be used in a SET STM statement.

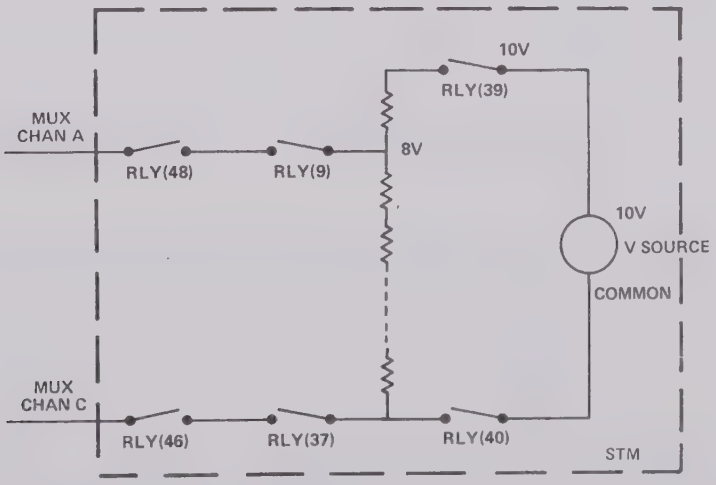


Figure 6-10. Sample STM Connections for Testing DC Voltmeter.

-------TPS (Triple Power Supply)-------

General

The system may contain up to 2 optional Triple Power Supplies (TPS). Each TPS has 3 independent power-supply sections that can be programmed to any 1 of 3 pre-selected voltage levels: VA, VB, and VC. These voltage levels are determined by programming resistors mounted on a connector that plugs into the TPS chassis. Section 4 of the Test Set Preparation manual contains more detailed information on the TPS.

Each TPS can be controlled by a SET, MEAS or CLEAR statement. A SET TPS statement controls the output of the designated TPS, by turning it on to 1 of its 3 pre-selected levels (VA, VB or VC). A MEAS TPS DEVTYPE reports on whether or not the designated slot contains a TPS. The CLEAR TPS command turns off the indicated TPS; note that the output voltage takes some time to decay to 0 V.

Format

```
SET TPS (slot) VA/VB/VC  DLY=delay;

MEAS TPS (slot) DEVTYPE INTO variable [error-command;];

CLEAR TPS (slot); or CLEAR TPS;
```

where:	SET/MEAS/CLEAR	Command keywords. Must appear first in the statement.
	TPS	Keyword for Triple Power Supply.
	(slot)	The TPSs can be inserted in slots 1-3 and 4-6. To specify a power-supply section of the TPS, use the number of the slot in which that supply is located (e.g., TPS(5)). Slot may be defined either as an integer (1-6) or a variable.
	VA/VB/VC	Keywords VA, VB and VC refer to the A-level, B-level and C-level outputs from each TPS section, as described in Section 4 of the Test Set Preparation manual. The actual voltage for each of these 3 levels is determined by programming resistors mounted on the rear of the TPS unit.
	DLY=delay	Optional parameter specifying additional time delay for for the TPS to settle. Refer to DLY description for UPS.
	DEVTYPE	Keyword used with a MEAS TPS statement to determine if a TPS is located in the specified slot.
	INTO variable	Defines the variable into which the device type indication is stored; a 1 indicates that a TPS is plugged into that slot; a 0 indicates no TPS.
	[error-command;]	A test language statement that is executed if an invalid TPS is specified or other problems are encountered when reading DEVTYPE.

Examples

1. SET TPS(5) VA [WRITE 'TPS(5) FAILURE %NL';];

   Meaning: Activate the TPS in slot 5 to output voltage level VA. If a failure is detected in the TPS output, display the error message.

2. MEAS TPS(3) DEVTYPE INTO DEVICE FAIL( ) [WRITE 'NO TPS(3) FOUND! %NL%';];

   Meaning: If slot 3 contains a TPS, store a 1 in DEVICE and execute the next instruction. If slot 3 does not contain a TPS, store a 0 in DEVICE, do not set a FAIL bit, and write the message: "NO TPS(3) FOUND!" on the screen.

Rules and Considerations

1. A CLEAR TPS statement does not isolate the TPS output from the UUT. A relay operated by the PIO board should be used for this purpose. (Refer to Section 4 in the Test Set Preparation manual.)

2. A CLEAR TPS statement with no slot specified clears all TPS units.

----------UPS (Universal Power Supplies)----------

General

Each UPS may be programmed over its full-scale voltage range (10-V, 20-V, 40-V and 80-V). In addition, the 80-V UPS may also operate in a current source mode, with output current programmed to any value within its range. The present 2270 system has 3 power-supply slots available for UPS (slots 4, 5 and 6). A 10-V, 20-V, or 40-V UPS requires 1 slot; the 80-V UPS requires 2 slots. Refer to Section 2 for more detailed information.

The supplies are programmable by means of several test language commands that are used to perform the following functions.

SET UPS forces (or attempts to force) the specified UPS to the indicated output. The 10-V, 20-V and 40-V UPS may be programmed to a specific output voltage with a specified current limit. The 80-V UPS may be programmed to a specific output voltage or output current with a limit set on the dependent variable. The specified limit on the dependent variable is independent of sign (e.g., a current limit of 1.5 A allows the UPS to source or sink up to 1.5 A).

TEST UPS compares the value of the dependent variable (e.g., in DCV mode it checks output current, in DCI mode it checks output voltage) against specified limits. If the tested value is within limits, the next statement is executed; otherwise, the error command is executed. An unsuccessful test stores the actual value in the variable defined by INTO; a successful test has no effect on that variable.

MEAS UPS measures and stores (in the variable specified by the INTO parameter) the value of the dependent variable. If INTO is not specified in a MEAS UPS command, the statement is internally converted to a TEST UPS command.

MEAS UPS DEVTYPE stores (in the variable specified by the INTO parameter) an indication of what power-supply type is in the specified slot.

CLEAR UPS resets the indicated power supply and forces its output to zero. It differs from a SET UPS V=0 command in that it does not wait for the supply output to settle to zero. As a result, it should not be used just prior to making a test or measurement. CLEAR UPS is typically used to reset all UUT power supplies at the end of a test sequence.

Format

```
SET DCV/DCI UPS(slot) V=voltage I=current DLY=delay
INTO variable FAIL(n)  [error-command;];

TEST/MEAS UPS(slot) limits INTO variable FAIL(n) [error-command;];

MEAS UPS(slot) DEVTYPE INTO variable [error-command;];

CLEAR UPS(slot);  or CLEAR UPS;
```

where: SET/TEST /MEAS/CLEAR — Command keywords.

DCV or DCI — Optional parameters that specify the function to be supplied. When used, it must appear before the keyword UPS (e.g., SET DCI UPS). Establishes whether the UPS will operate as a voltage source or current source. Only the 80-V UPS can operate as a current source (DCI). The 10-V, 20-V and 40-V UPSs can only operate as a voltage source (DCV). The default function is DCV.

UPS(slot) — Keyword for Universal Power Supply followed by the slot location. UPSs can only be inserted in slots 4, 5 and 6.

V=voltage — Required parameter for SET UPS. Specifies output voltage for the UPS listed. In voltage mode (DCV), the supply output is forced to a level within the voltage range shown below in the steps indicated. In the current mode (DCI), output voltage is limited to this value. If the specified value is outside the range of the unit, then a failure is indicated.

Power Supply	Voltage Range (V)	Steps (mV)
10 V	-10.24 to +10.235	5
20 V	-20.48 to +20.47	10
40 V	-40.96 to +40.94	20
80 V	-81.92 to +81.88	40

I=current — Required parameter for SET UPS. Specifies the output current for the UPS listed. In voltage mode (DCV), the output current is limited to this value (if it is within the current range of the supply). In current mode (DCI), the output current is forced to this level if it is within the current range and resolution of the unit as follows:

Power Supply	Current Range (A)	Steps (mA)
10 V	-4.096 to +4.088	8
20 V	-2.044 to +2.048	4
40 V	-1.024 to +1.022	2
80 V	-0.512 to +0.51175	0.25

DLY=delay — Optional parameter. Specifies additional time needed by the power supply to settle, allowing for stored charge in the UUT. Maximum = 2 seconds. At present, to determine if the power supply has settled, the output is checked as follows:

1. After 2 ms plus the value of DLY, the output is checked.

2. If the output has not settled by this time and the interval is less than 30 ms, this output is again checked 30 ms after the SET UPS command was issued. If the supply has not settled by then, a failure is indicated.

limits — Limits are an integral part of the TEST UPS statement but are also allowed in a MEAS UPS statement. They may be expressed as follows:

HI= (Single-sided) or
LO= (Single-sided) or
HI= LO= (Double-sided) or
VAL= TOL= (Nominal value with tolerance expressed as a percentage).

INTO variable — In a SET UPS statement, it specifies variable name into which the system will store a "1" if UPS has failed to settle.

In a TEST UPS statement, this parameter is optional. If used, it specifies the variable in which the system will store the actual value if the tested value exceeds the specified limits.

In a MEAS UPS statement, this parameter must be used to indicate the variable in which the measured value is stored. In a MEAS DEVTYPE statement, this parameter specifies the variable into which the following UPS type indication will be stored:

0 - NO UPS
10 - 10-V UPS
20 - 20-V UPS
40 - 40-V UPS
80 - 80-V UPS

FAIL(n) — Optional FAIL bit. See Fail vector description in para 5.2.

[error-command;] — The command(s) within these brackets is executed whenever the statement cannot be performed. In a SET UPS statement, a failure occurs if the supply does not regulate in the allotted time or if it is not capable of producing the specified output. Examples of this latter failure include

specifying an out-of-range output (e.g., V=15 from a 10-V UPS), or specifying a very low output where the resolution of the supply introduces excessive error (e.g., 10 mV from a 40-V UPS, where the resolution is 40 mV per step). In a TEST UPS statement, a failure occurs if the supply is outside of the specified limits. In a MEAS UPS statement, a failure occurs when the measurement cannot be made. In a MEAS DEVTYPE statement, a failure occurs if no UPS is found in the indicated slot.

DEVTYPE — Keyword used in a TEST UPS statement to indicate that the UPS type is being requested and not the actual value of the UPS output.

Examples

1. SET DCV UPS(4) V=8.5 I=1.5 INTO PS4 FAIL(7) [CALL UPSERR;];

   Meaning: Set the UPS in slot 4 to 8.5 V with a current limit of 1.5 A. If the supply output fails to settle, store a 1 in variable PS4, set FAIL bit 7 and call subroutine UPSERR.

2. TEST UPS(4) VAL=1.0 TOL=10 INTO TSTERR FAIL(7) [CALL ERR;];

   Meaning: Check that the output current of UPS(4) falls between 0.9 and 1.1 A. If it doesn't fall within these limits, store the actual value in TSTERR, set FAIL bit 7 and call subroutine ERR.

3. MEAS UPS(4) INTO VAL;

   Meaning: Measure the output current of UPS(4) and store the value in VAL.

4. MEAS UPS(4) DEVTYPE INTO T;

   Meaning: Store in variable TYPE an indication of what type UPS is in slot 4. A 0 indicates no UPS in that slot; 10, 20, 40 and 80 indicate 10 V, 20 V, 40 V and 80 V, respectively.

Rules and Considerations

1. UPSs can only be placed in slots 4, 5, and 6.

2. To specify a UPS, use the slot number in which the supply is located (e.g., UPS(4)). The 80-V UPS, which requires 2 slots, is identified by the first slot number (e.g., an 80-V UPS in slots 5 and 6 is identified as UPS (5)).

3. A SET UPS to 0 V statement or a CLEAR UPS does not isolate the UPS output from the UUT. A relay operated by the PIO board should be used for this purpose. Refer to Section 4 of the Test Set Preparation manual.

4. In a 10-V, 20-V or 40-V UPS, a FAIL condition occurs if the output current exceeds the limit set by the I= parameter.

5. A CLEAR UPS statement with no slot specified will clear all UPS units.

6.4 DIGITAL TESTING

6.4.1 General

An introduction to the general principles of digital device testing on the 227X is presented in para 4.3. Main features of the digital test system are summarized in the following paragraphs.

Digital ICs are tested by inputting logic voltage pulses to the IC under test, with sufficient drive capability to force each input to the desired state regardless of any interfering signals. This "backdriving" technique is illustrated in Figure 6-11 by two interconnected type 7400 NAND gates.

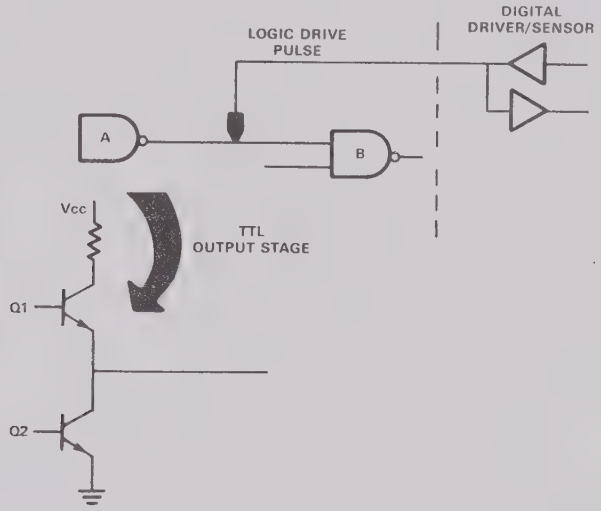


Figure 6-11. Backdriving Digital ICs.

In Figure 6-11, a low (e.g., 0 V) output at gate A indicates that transistor Q1 in the output stage of that IC is turned on (i.e., in a low impedance state). In order to overcome this condition and force the input to gate B high (e.g., +4 V) for testing, a drive pulse is applied to the gate A output, with sufficient drive current to develop the required voltage across Q1. In typical applications, a current of 150 to 400 mA may be required to back-drive gate A and obtain a +4 V level at gate B. The drive pulse is kept short enough to prevent excessive power dissipation in the IC, but long enough to test the IC outputs for proper operation.

Each digital (or hybrid) test nail is connected to a driver and sensor circuit, each of which is separately controlled by the test program. The drivers develop the logic inputs (stimuli) and the sensors monitor the logic outputs (responses). An internal protection circuit in each driver automatically reduces the drive current to tolerable levels if the driver remains on too long, thus protecting the UUT from damage.

A 1K memory controls each driver/sensor nail and allows digital testing to be performed in a high-speed burst. During a burst, the test program segment for an IC is preloaded in the driver/sensor memory; then, the test is executed at high-speed. During this burst, sensor data is stored "on-the-fly" in the driver/sensor memory where it is available for further analysis after the burst is complete. The output of each driver/sensor also contains a current source

for shorts testing of digital nodes, and a pair of pull-up and pull-down resistors for testing tri-state devices.

Systems having the optional SpeedPlus digital test hardware can program the SpeedPlus clock generator module to supply programmed clock waveforms to special clock driver nails. These signals can be synchronized to a UUT signal sensed at a special synchronization sense nail. During execution of a BURST section, program control can be transferred to the SpeedPlus subsystem by a FAST statement within the burst. Subsequent test statements will then execute at higher speed, in synchronism with the clock module. Moreover, the program flow can be aligned with bus events, such as the read/write cycles of a microprocessor, using SpeedPlus trigger nail parameters. TEST LGC and SET logic statements within the FAST section can, however, only reference nails that are assigned to the SpeedPlus subsystem.

Each driver/sensor has a relay in series with its output, used to connect the driver/sensor to the digital (or hybrid) test nail. The user has no direct control over this relay. At the start of each burst, the system automatically closes the relays attached to the test nails used in that burst and automatically opens the relays at the end of the burst, unless otherwise instructed by an ACTIVE specification in the BURST statement.

The following example illustrates the overall structure of the digital portion of a 227X test program.

```
DECLARE <nail groups, nailbinding>  /*DECLARATIONS*/
CLEAR MUX;                          /*OPENS ALL MUX RELAYS*/
CLEAR SCAN;                         /*OPENS ALL SCAN RELAYS*/
SET PIO(0) HRLY (CLOSE 3);          /*APPLIES SYSTEM GROUND TO UUT*/
SET PIO(0) HRLY (CLOSE 7);          /*CONNECTS UPS OUTPUT TO UUT*/
SET DCV UPS(4) V=5;                 /*TURNS ON POWER SUPPLY*/
DELAY 10M;

ASSIGN LGC VIHA=4.5 VILA=0.0
  VOHA=2.1 VOLA=1.0 LVLA(1-3);      /*DEFINES LOGIC LEVELS AND TEST NAILS*/

SET CLOCK <clock waveforms, timing data> /*SpeedPlus CLOCK MODULE*/
          '
          '
Digital bursts:

  1.  Defining the burst:      BURST/END BURST

  2.  Defining SpeedPlus high-speed sections:  FAST/END FAST

  3.  Testing the logic:       SET LGC, TEST LGC

  4.  Looping:                 LOOP/END LOOP, LOOPIF, EXITIF, OR

  5.  Branching:               GOTO, GOSUB, SUBROUTINE/RETURN/
                               END SUBROUTINE, BREAK
          '
          '
CLEAR LGC;                          /*RESETS ALL DIGITAL LOGIC*/
          ¦
```

```
CLEAR UPS(4);                          /*TURNS OFF POWER SUPPLY*/
SET PIO(0) HRLY(OPEN 7);               /*DISCONNECTS UPS FROM UUT*/
SET PIO(0) HRLY (OPEN 3);              /*REMOVES SYSTEM GROUND FROM UUT*/
CLEAR  CLOCK;                          /*CLEAR SpeedPlus CLOCK MODULE*/
```

After analog testing is complete, the MUX and SCAN relays should be opened before starting digital testing. Power and system ground must be applied to the UUT prior to digital testing and removed before using the SCANner or MUX for hybrid testing. The SET PIO and SET/CLR UPS (or TPS) statements can be used to accomplish this. Also, logic levels must be established for both the digital drivers and the logic sensors, and test nails must be defined prior to any digital burst (see ASSIGN LGC). Once digital testing is complete, a CLEAR LGC statement may be used to reset the digital driver/sensor system. Neither ASSIGN LGC nor CLEAR LGC can be used within a burst.

The instructions that can be used within a burst are listed in Section 3. The boundaries of a burst block within a test program are defined by a BURST statement at the beginning and an END BURST at the end. Driver/sensor control of the UUT nails is accomplished by SET LGC and TEST LGC statements, which define all nail specifications within the burst.

For added flexibility, special control statements are allowed in a burst. These differ from the control statements used outside of bursts. GOTO is used for either an unconditional transfer or a conditional transfer, based on the pass/fail status of the previous test. GOSUB is used to call a subroutine either conditionally or unconditionally and to return program control to the main flow once the subroutine is executed.

The SUBROUTINE and END SUBROUTINE statements identify the first and last statement of a subroutine within a burst. A RETURN statement must immediately precede the END SUBROUTINE statement to cause a transfer back to the main program. (Subroutines are not allowed in FAST sections of a test and FAST sections are not allowed inside subroutines.)

Looping within a burst is accomplished by the LOOP/END LOOP statements. Additional looping or exit conditions can be specified via LOOPIF, EXITIF and OR modifiers.

BREAK is used to force an early exit from a burst and to specify the state of the hardware after the exit.

FAST is used only on systems having the SpeedPlus digital testing option to enter a high-speed section of a burst. END FAST signals the end of the high-speed test steps; any remaining steps are executed at normal speed until either another FAST statement or an END BURST statement appears.

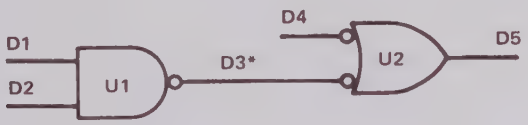
6.4.2 Power and Ground Wiring

During in-circuit testing of analog components, no power is applied to the UUT and UUT ground is not connected to the test system ground. To perform in-circuit testing of digital ICs, however, both power and ground must be applied to the UUT. Program control of relays via the Relay Driver boards (PIO) simplify the connecting and disconnecting of user power supplies and functional ground to the UUT.

Refer to the Test Set Preparation manual for a description of the relay drivers and user power supplies and for guidelines on wiring UUT grounds. Refer to para 6.3, Analog Instrumentation Statements, for the test language statements used to control the relay drivers (PIO) and power supplies (UPS/TPS).

6.4.3 Digital Test Nail Commands

The LGC statements control all digital inputs and monitor all digital outputs on the UUT. The drivers provide the input stimulus to the UUT and the sensors check the response. Each digital nail in the system is physically connected to a logic driver and to a sensor circuit. Under program control, a driver can be electronically disconnected from a test nail and a sensed output from a nail can be independently ignored. As a result, the same nail may be used as an input for one test or an output for another test (see Figure 6-12).



*Test nail D3 is used as an output nail when testing U1 and as an input nail when testing U2.

Figure 6-12. Sample Logic Circuit Test Connections.

In general, the LGC statements are used to reset the hardware, define the logic voltage levels, connect/disconnect the drivers, monitor or ignore the sensor outputs, and designate the logic state of each nail. The LGC statements consist of the following commands:

ASSIGN LGC	Defines the voltage levels for drivers and sensors (typically used once per program).
SET LGC	Drives all inputs to their specified state, but performs no testing.
TEST LGC	Drives all inputs and tests the results.
CLEAR LGC	Resets the driver/sensor system.

The ASSIGN LGC, SET LGC, and TEST LGCcommands all have nail specifications (nail-specs) associated with them. These nail-specs allow the user to assign the logic families (ASSIGN LGC) and to define the driver/sensor condition (input, output, connect, disconnect, ignore, sense) and the logic state (high, low, mid-state or unknown) of all nails in the system. Except in an ASSIGN LGC statement, each nail-spec consists of one or more entries, each consisting of a 2-character nail state designator, followed by a nail list. For example, IH(1-3, 7, 9) causes input nails 1, 2, 3, 7, 9 to be driven high. Table 6-3 shows the list of allowable nail-state designators.

Table 6-3
DIGITAL NAIL STATE DESIGNATORS

Inputs

IC	Input Connect	Enable input logic driver
ID	Input Disconnect*	Disable and place driver in a high-impedance state
IL	Input Low*	Drive input nail low
IH	Input High	Drive input nail high
II	Input Ignore	Ignore this input
IS	Input Sense	Enable sensing of these inputs
IG	Input Group	Group of input nails assigned by a DECLARE GROUP

Outputs

OL	Output Low*	Expected output state is low
OH	Output High	Expected output state is high
OM	Output Midstate	Expected output is between high and low thresholds. (not permitted on SpeedPlus nails)
OI	Output Ignore*	Ignore these outputs
OS	Output Sense	Enable sensing of these outputs
OU	Output Unknown	Output state currently unknown; to be learned via LEARN mode (refer to Test Set Preaparation manual)
OG	Output Group	Group of output nails assigned by a DECLARE GROUP statement
TH	Trigger High	Trigger on highs at these SpeedPlus inputs
TL	Trigger Low	Trigger on lows at these SpeedPlus inputs

* State of all active nails after a CLEAR LGC, at the beginning of a burst, or at the start of every subroutine.

Terminating Resistors

PU	Pull-up only
PD	Pull-down only
PB	Both pull-up and pull-down
PN	Neither pull-up nor pull-down (default condition)

The terminating resistor connections to the driver/sensor nails are used mainly for testing bus structures. A 3.3-K pull-up resistor connects to the driver high reference level (VIH in the ASSIGN LGC statement), and a 2.2-K pull-down resistor connects to the driver low reference level (VIL in the ASSIGN LGC statement). These terminating resistor specifications can only be used in a SET LGC statement that contains no other nail specs or parameters. An example using these resistors to test a tri-state bus is given in the description of the SET LGC statement.

Format

The 4 LGC statements (i.e., ASSIGN LGC, CLEAR LGC, SET LGC, TEST LGC) are described in detail as they appear in alphabetical order in this test language description.

Rules and Considerations

1. Individual nail numbers in a nail list are separated by commas (e.g., 1,3,5).

2. A "through" construct is allowed using a hyphen (-) to indicate a series of sequential nails (e.g., 1-4 means nails 1 through 4). In all LGC statements, this "through" construct must specify the series of nails in ascending order (e.g., 1-4, not 4-1). In a DECLARE GROUP statement, however, the nail sequence must be specified in whatever order is necessary to define the group properly.

3. With the exception of the "through" construct described in item 2, a LGC command nail list may be specified in any order, for example IH(1-3, 9, 7). In a DECLARE GROUP statement, however, where the order of the nails defines a binary number, the "most significant" nail must appear first in the list and the "least significant" nail last. Thus, in a DECLARE GROUP statement, NAME (1-3, 7, 9) and NAME (9, 3-1, 7) are both allowable nail lists but care must be taken to use the appropriate one.

4. Nails are considered active or inactive in a nail specification depending on the type of specification and where it appears in the program. The definition of active nails for each nail specification is as follows:

Nail Spec	Active Nails
LVLA, LVLB (part of ASSIGN LGC)	All of the nails allowed in the maximum system configuration and not explicitly referenced by the parameter.
IC, ID	All nails that have been assigned (i.e., all nails referenced in the most recent ASSIGN LGC statement, in compile-time order).
IS, II, IH, IL	All nails that are currently specified as connected (IC).
OS, OI	All nails that have been assigned but not currently connected.
OH, OL, OM, OU	All outputs that are currently being sensed (i.e., all nails that are assigned, not currently connected, and specified as being sensed).

5. The # symbol may be used to indicate an "all-but" condition in a nail list (i.e., all active nails but those specified). For example,

   IH(#3,4)

   means that all active inputs are high except ("all but") nails 3 and 4. By itself, means "all": for example, ID( ) means disconnect all outputs.

6. Until otherwise specified, all nails are assumed to be disconnected, ignored, in the low state, and with no terminating resistors.

7. With the exception of the trigger nails in the SpeedPlus option, nail specifications are incremental, that is, a nail remains in its present state until explicitly redefined in a program statement. These nail

specifications are incremental in the order in which they appear in the program listing (at compile time), but not necessarily incremental in the order in which the program executes them. There is a distinct difference between the two if the program executes a branch around statements that modify nail states. The following simple example illustrates the difference.

Statement	At Compile Time (Actual nail states)		During Execution (Expected states if nail-specs were run-time incremental)	
	High Nails	Low Nails	High Nails	Low Nails
IH(1-3)	1,2,3	**	1,2,3	**
IL(2)	1,3	2	1,3	2
GOTO 10	1,3	2	1,3	2
IH(2)*	1,2,3	**	*	*
X10: IL(1,3)	2	1,3	**	1,2,3

* Not executed.
** No nails in this state.

8. Once a nail state is specified as output unknown, that OU specification must be carried through all subsequent test steps, as long as the nail is active.

9. The compiler does not check for nail-spec inconsistencies of the type IL(1) IH(1). In such cases, the final specification (IH(1) in this example) prevails. In other cases, where an attempt is made to define the same nail as both an input and an output, for example, an input specification (IS, II, IH or IL) to a disconnected (ID) nail or an output specification (OL, OH, OM, etc) to a connected IC nail, the compiler will issue an error message.

10. Some nail-specs imply others; for example, input connect (IC) implies input sense (IS), and input disconnect (ID) implies input ignore (II).

11. In some cases, the order is important. For example, an input nail must be disconnected before it can be specified as an output nail, -- ID(2) OS(2), not OS(2) ID(2). Similarly, CI(3) IC(3) is acceptable, but not IC(3) OI(3).

12. OM states are stored but are not tested by the hardware during a burst. As a result, the OM state should not be specified as a condition for branching and looping, since it will be ignored during the burst.

13. ATG-produced test programs create inputs to a data-base used by the diagnostic system. These inputs include descriptive labels which the diagnostics use in failure analysis. Thus, it is highly recommended that the user not change or include new [error-branches] in ATG-produced test programs.

14. For convenience in writing test programs, nails may also be specified in groups. For example,

    IC(OPCODE) will connect drivers to all UUT nails belonging to the nail group defined as OPCODE.

PB(BUS) will connect both a pull-up and a pull-down resistor to all nails in the nail group defined as BUS.

IG(ADDRESS=O'15') will set the nails in input nail group called ADDRESS to an octal 15. The order of the nails is defined in the DECLARE GROUP statement.

Refer to the description of the DECLARE GROUP statement for details on nail group specifications.

6.4.4 Test Nail Numbering

Test language statements identify a nail both by type (digital, hybrid, SpeedPlus clock, etc.) and by number. Because a 227X system can be configured with varying numbers of test nails, it behooves the programmer to ensure that only those test nails are used that are physically present in the system. (Programs moved from another test system will need to be revised if they reference unavailable hardware.) Listed below are the types of nails and their maximum ranges for various 227X systems. Depending on the system configuration, some nail ranges may not be continuous; i.e., some nails within the range may be missing.

2270

The 2270 can be configured with nails for digital testing only, for analog testing only, or for both digital and analog testing, or various combinations of these. Nails that can be used for both analog and digital tests are called hybrid nails.

Type and Designation	Maximum Range
Digital test nail (D)	D1 to D960
Analog test nail (A)	A1 to A1024
Hybrid test nail (H)	H1 to H480

Other 227X

Other 227X systems are supplied only with hybrid test nails, at which both digital and analog tests can be performed.

Type and Designation	Maximum Range
Hybrid test nail (H)	2271: H1 to H960 2272: H1 to H3584 2275: H1 to H640

SpeedPlus Option (2271, 2272 Workstations)

Systems having the SpeedPlus option have a number of additional nails available for executing high speed digital tests. These fall into 4 categories: clock driver, clock synchronization, trigger, and high-speed hybrid test (driver/sensor) nails. The first 3 categories, referred to as timing nails, serve auxilliary functions in SpeedPlus testing (usually in the FAST section of a burst). The test nails, however, can be used both in standard analog and digital tests, as well as in FAST sections. Hence any part of a test program

can reference nails F4001 and above, while only those portions of the program that use the SpeedPlus hardware (mainly CLOCK statements and FAST sections of bursts) will refer to the timing nails (types CD, CS, and T).

Type and Designation Range

Clock Driver A (CDAn) n = 1 to 16
Clock Driver B (CDBn) n = 1 to 16
Clock Driver C (CDCn) n = 1 to 16
Clock Driver D (CDDn) n = 1 to 16

Clock Sync (CSn) n = 1 to 8

Trigger (Tn) n = 1 to 26, but excluding
T23, T24 (2271 only)
Trigger (Tn) n = 1 to 28 (2272 only)

Hybrid Test (Fn) 4001 to 4384 (2271 SpeedPlus only)
Hybrid Test (Fn) 4001 to 4864 (2272 SpeedPlus only)

6.4.5 Digital Test Programming Statements

The 227X test language statements unique to digital testing are presented in the following paragraphs, IN ALPHABETICAL ORDER. These statements consist of:

ASSIGN LGC	GOTO
BREAK	LOOP/END LOOP
BURST	LOOPIF
BUSTEST/END BUSTEST	RETURN
CLEAR LGC	SET LGC
EXITIF	SUBROUTINE/END SUBROUTINE
GOSUB	TEST LGC

Systems with the SpeedPlus digital testing option will, in addition, recognize the following statements:

CLEAR CLOCK	GOTO FLAG PASSES
DECLARE NAILBIND	GOTO NOPHASELOCK
FAST/END FAST	GOTO PHASELOCK
FLAGFAIL	SET CLOCK
GOTO FLAG FAILS	TEST CLOCK

In FAST sections the TEST LGC and SET LGC statements can also take the additional trigger parameters TH, TL, and FLAGFAIL; they cannot take OM.

The BURST and ASSIGN LGC statements can take the VTT parameter; ASSIGN LGC also recognizes parameters VCDL, VCDH, VCST. GOSUB is not recognized in FAST sections.

----------ASSIGN LGC----------

General

This instruction defines the logic voltage levels to be used in subsequent tests and assigns these levels to the appropriate UUT nails. Two independent sets of voltages (referred to as sets A and B) may be specified. Each set contains 4 separate voltages: VIH and VIL define the high and low logic driver levels (UUT inputs), and VOH and VOL define the high and low sensor threshold voltages. The logic drivers provide the input stimulus to the UUT, and the logic sensors monitor the UUT outputs. Each voltage is independently programmable between its upper and lower source limits, as defined in the Value Specifications paragraph below.

Format

```
ASSIGN LGC  VIHA=  VILA=  VOHA=  VOLA=  LVLA(nail-list)
            VIHB=  VILB=  VOHB=  VOLB=  LVLB(nail-list)
            VCDL=  VCDH=  VCST=  VTT=;
```

where:

ASSIGN LGC — Command keywords. Must appear first in the statement.

VIHA= .... VOLA= — The 4 voltage specifications for set A.

LVLA(nail-list) — List of nails assigned to voltage set A.

VIHB= .....VOLB= — Optional. The 4 voltage specifications for set B.

LVLB(nail-list) — Optional. List of nails assigned to voltage set B.

SpeedPlus Option Only

VCDL= ... VCDH= — Used to define the low and high logic voltages applied to the UUT by the SpeedPlus clock driver circuits. Defines the default values used by SET CLOCK.

VCST= — Defines the low-high logic threshold for the synchronization input from the UUT to the SpeedPlus clock module. Defines the default values used by SET CLOCK

VTT= — Defines the low-high logic threshold for the trigger inputs from the UUT to the SpeedPlus triggering module. Defines the default threshold for a BURST section.

Examples

```
1.  ASSIGN LGC  VIHA=3.1  VILA=.5  VOHA=2.9
VOLA=.8    LVLA(1,31-39)
                     VIHB=.5   VILB=-4  VOHB=.0
VOLB=-3.5  LVLB(2-30,40);
```

Meaning: Set logic level set A to the designated voltages (3.1 V, .5 V, etc) and logic level set B to its designated voltages (.5 V, -4 V, etc).

Nails 1 and 31 through 39 use the A-voltages and nails 2 through 30 and 40 use the B-voltages.

2. ASSIGN LGC VIHA=2.5 VILA=.4 VOHA=2.2 VOLA=.0 LVLA (1-20);

   Meaning: Set logic level set A to designated voltages 2.5 V, .4 V, etc. All active nails will use these A-voltages.

Rules and Considerations

1. Logic voltages defined by an ASSIGN LGC statement remain in effect throughout the whole program or until specifically changed by another ASSIGN LGC statement.

2. An ASSIGN LGC statement does not assign nails incrementally (i.e., each ASSIGN LGC statement cancels all nail assignments made by the previous ASSIGN LGC statement). Therefore, when using both voltage sets (A and B), always use a single ASSIGN LGC statement to assign both sets of nails.

3. If a logic family is referenced, all 4 voltages and the nail list for that family must be specified.

4. Nails that are not included in an LVLA( ) or LVLB( ) nail list cannot be used in the burst.

5. The ASSIGN LGC statement must be executed before any other LGC statement in a program.

6. An ASSIGN LGC statement cannot appear within a burst block.

7. The ASSIGN LGC statement preceding a burst in compile-time order determines the active nails for # ("all-but") processing within the burst. (Refer to para 6.4.3.)

8. ASSIGN LGC instructs the compiler how to set up the voltage levels for the following BURST. Thus, branching around an ASSIGN LGC has no effect.

9. In typical operation, the driver levels should be "greater" than the sensor thresholds. See Figure 6-20 in the TEST LGC description. The compiler issues no warning if high thresholds are lower than low thresholds.

10. Inputs are sensed at the assigned output voltage level. As a result, if the output level specified is equal to or greater than the input level, the input will fail when sensed.

Value Specifications

1. All voltages are programmable between -6 V and +10 V.

2. Variables cannot be assigned as values for VIHA, VILA, etc.

----------BREAK----------

General

This statement is used to force an early exit from a burst and to specify the state of the hardware after the exit. Following the BREAK, program control is transferred to the first statement following END BURST.

Format

```
BREAK CLEAR/ACTIVE/null;
```

where:

BREAK	Command keyword.
CLEAR/ACTIVE/null	Specifies the state of the hardware after exiting a burst. Overrides the CLEAR/ACTIVE specification associated with the burst block. A null causes the CLEAR/ACTIVE specification of the associated burst to be used.

Examples

1. BREAK ACTIVE;

   Meaning: Immediately exit the burst and leave the driver/sensor system and all nails in their existing state.

2. BREAK CLEAR;

   Meaning: Immediately exit the burst and reset the hardware and all test nails to the CLEARed state (disconnected, ignored and low).

Rules and Considerations

1. BREAK can only be used in a burst.

2. BREAK in a FAST section causes a branch to the end of that section.

----------BURST/END BURST----------

General

A burst consists of a clearly-defined group of statements that define each step taken by the driver/sensor system operating in a high-speed mode. Each burst typically tests one IC or, in the case of more complex MSI or LSI chips, each burst may test a portion of an IC. The object code for a burst is stored in a 1K RAM in the driver/sensor (D/S) system, which can typically store more than 400 test vectors.

Once the burst is stored in the D/S memory, the D/S system then begins a high-speed execution of the test. Test results are stored "on-the-fly" in the D/S memory and analyzed after the burst is complete.

A burst block is typically assigned a label for reference. If no label is explicitly assigned, no digital diagnostics will be associated with that section.

Variables defined in a burst are only available within that burst. Refer to "Rules and Considerations" for information about the interaction of BURST and analog instrumentation statements.

SpeedPlus Subsystem

Systems having the SpeedPlus testing option recognize an additional set of BURST parameters that are directed specifically to the high-speed test hardware. However, only those test statements within the burst that are bracketed by the statements FAST and END FAST will actually execute as a SpeedPlus test, using the higher clock rates and synchronization features of the SpeedPlus hardware. Other test statements not within a FAST section may reference nails that belong to the SpeedPlus option, but they execute exactly as all other standard test statements. Any number of fast sections can be interleaved with standard test sections.

Note that timing for fast and standard sections of a burst is defined in 2 quite different ways. The parameters IST and ITT, which are part of the BURST statement, define the test rate as well as drive and sense times for all non-fast statements in the burst. Timing for the fast sections is not defined in the BURST statement. All fast sections are instead controlled by the most recent SET CLOCK statement, which is not part of the burst. (This allows the clock drivers to be activated independently of burst activity if necessary, as when dynamic memory devices must be kept 'alive' from one burst to the next.)

When FAST sections are synchronized to the UUT, it is usually desirable to monitor whether or not synchronization is maintained. In some situations, however, synchronization will be lost deliberately, so monitoring should be disabled with the keyword NOSYNC.

Format

```
label:  BURST  IST=value  ITT=value  MAXTIME=value CLEAR/ACTIVE
               BUS NAME NOPRINT NOPROBE INTO variable
               VTT=value fail-opts FAIL(n) SYNC ¦ NOSYNC
               [error-branch;];
                  '
               <std test steps, e.g., IC(n) OS(m); >
                  '
               FAST;
                  '
                  <fast test steps, e.g., TL(n) IH(4nnn); >
                  '
               END FAST;
                  '
               <more std steps or fast sections>
                  '
        END BURST [error-commands;];
```

where: label — Provides a reference name for the burst. If not labelled, no digital diagnostics will be available to that burst.

BURST — Keyword indicating the start of the burst.

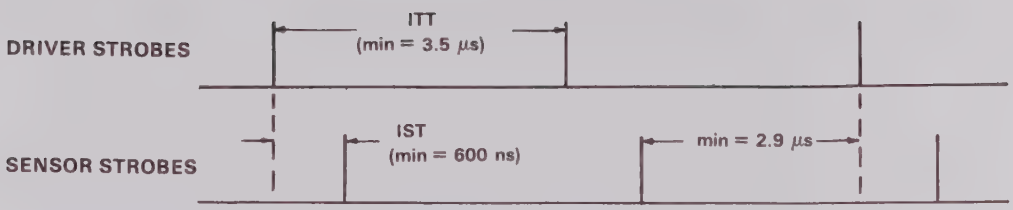


Figure 6-13. IST/ITT Timing.

IST= value and
ITT= value

Keywords specifying Interstrobe Time (IST) and Inter-Test Time (ITT) in seconds. IST is used to define the time interval between driving the input nails and sensing the output nails in a test. ITT is used to define the time interval between consecutive tests. The timing diagram in Figure 6-13 shows the relationship between the IST and ITT values specified in the BURST statement and the actual driver and sensor strobes in the test system.

The following rules apply:

1. The minimum time interval allowed by the hardware between driver strobe and sensor strobe is 600 ns. Thus, if IST is specified as 100 ns by the BURST statement, the actual time interval will default to 600 ns.

2. The minimum time interval between driver strobes is 3.5 us. If ITT is specified as less than this, the actual value will default to 3.5 us.

3. The difference between ITT and IST (i.e., ITT - IST) must be greater than 2.9 us.

Two ranges (not controllable by the user) are available for both IST and ITT:

0.1 to 25.6 us (0.1-us resolution)
0.5 to 128.0 us (0.5-us resolution)

Range is automatically selected by the system, based on the value specified. If the specified value falls outside the range or resolution of the system, it will be rounded-off to the nearest value within the system capability. If no values are specified,

IST = 1 us (default)
ITT = 4 us (default)

If IST and ITT are specified so that the values fall in 2 different ranges, the highest range (worst resolution) will be used for both. Note that the IST/ITT values are always cleared at the end of a burst.

MAXTIME=value — Keyword specifying the maximum time, (in seconds) allowed for executing the burst and the value in seconds may be any of the following values:

10 ms (10M), 40 ms (40M), 160 ms (160M), or 640 ms (640M).

The default value is 640 ms. If a time-out occurs, the burst is immediately exited and a fail condition is produced. Standard test language scale factors such as U (micro), M (milli), etc are allowed.

CLEAR/ACTIVE — Keywords specifying the state in which to leave the hardware upon completion of the burst. CLEAR resets the driver/sensor system and places all nails in the "off" state; it may also open analog scanner connections. ACTIVE leaves all nails in their existing state at the end of the burst. The default condition is CLEAR.

BUS — Optional parameter used to inform the system that this burst is being used to test a bus rather than an IC. If a bus test fails, most of the ICs connected to it will fail their tests. To prevent this, the BUS parameter causes the run-time system to check the .IDD file (refer to Test Set Preparation manual, Section 7) and remember all ICs connected to a failed bus, and then to skip over those IC tests whenever they are encountered in the test program.

NAME=' ' — Optional parameter used to provide RTS with the names of all ICs that are tested within a single burst (parallel testing). Using the NAME parameter provides data the run-time system needs to indict the IC as well as the failing nails in a parallel IC test burst.

NOPRINT — Optional parameter used to suppress all system diagnostic messages caused by a failure in the burst.

NOPROBE — Optional parameter used to inhibit probing the IC if it fails during the burst.

INTO variable — Optional parameter that specifies a bit string variable into which test nail failures are stored after the burst is complete. Each bit position in the variable represents a test nail; bit 1 corresponds to nail 1, bit 2 corresponds to nail 2, etc. A 1 indicates a nail failure, a 0 indicates no failure. If the bit-string variable is not long enough to handle the nail-list, a warning message will be printed and the system will attempt to continue with the burst, and ignore the missing bit positions.

VTT=value — VTT specifies the trigger threshold voltage. The permissible range is -6 to +12 volts. Levels above VTT are consider logic high; below it are considered logic

low. A VTT value must be defined if any TH or TL parameters appear in the burst; it may, however, be previously defined in an ASSIGN LGC statement. The burst parameter overrides the ASSIGN LGC value for this burst only.

fail-opts — A BURST may also take any of the failure reporting options described in para 5.3 (e.g., FSUB).

FAIL(n) — Optional. Fail vector bit n. Set to 1 after a burst if a failure occurs within the burst. If n is a null (i.e., FAIL( )), then no fail bit is set. If FAIL(n) is not specified at all, then fail bit 1 is set if a failure is detected.

NOSYNC ¦ SYNC — Unless the keyword NOSYNC is present in the BURST header, the following verification is made:

A. At the beginning of every BURST which contains FAST sections, the system waits for clock synchronization. If no synchronization is achieved the BURST exits as FAILED and writes to the MESFILE the message, "NO SYNCHRONIZATION ACHIEVED".

B. At the end of every BURST which contains FAST sections, the system checks if synchronization was maintained throughout the BURST. If synchronization is lost at any time during the BURST this will be reported in the MESFILE with the message - "SYNCHRONIZATION LOST"

If the NOSYNC is present in the BURST the above checks will not be made.

Default: SYNC

[error-branch;] — Optional parameter in a BURST statement. Specifies where to branch after the burst if a failure was detected within the burst. Must be the last entry in the statement and must be a BRANCH command. Cannot be used if END BURST contains an [error-command;].

[error-command;] — Optional parameter in an END BURST statement. Executed if the burst fails. May be any number of statements including another burst. Cannot be used if the BURST statement contains an [error-branch;].

std test steps — May include any of the following types of statements: SET LGC, TEST LGC, GOTO, GOSUB, RETURN, SUBROUTINE/END SUBROUTINE, BREAK, or statement modifiers LOOP, END LOOP, LOOPIF, EXITIF, OR.

FAST; — This statement marks the start of a section of test statements that execute on the SpeedPlus subsystem. Test statements in FAST sections may mention only test and timing nails that belong to the SpeedPlus option. During the FAST sections, other (standard) test nails

remain in their pre-FAST states. (I.e., standard nail context is preserved across the FAST section.) Timing for the FAST section is governed by the most recently executed CLOCK statement. Nail context for the standard test nails is preserved across the FAST section. Standard driver nails continue to be driven at the logic levels they have upon entry to the FAST section. All standard sensor nails, however, are disconnected during the FAST section; sensing is restored at the first test step after END FAST;. Any number of FAST sections (storage permitting) can be created within a burst.

fast test steps — May include any of the standard types of digital test statements, as well as FLAG PASSES, FLAG FAILS, NOPHASELOCK, PHASELOCK. They may not include GOSUB, GOTO PASSES, or GOTO FAILS. SET LGC ($) and TEST LGC can include the trigger terms TH, TL in addition to other logic test terms (e.g., IH, OS, etc.); they may not include OM. EXITIF may not appear within 5 steps before the end of a loop. Branches into, out of, or between FAST sections are not allowed.

END FAST; — This statement marks the end of a FAST section and returns program flow to the standard test system. Test nails in the SpeedPlus subsystem remain in the state they are in at the end of the FAST section; other test nails retain their states across the FAST section. Consequently, for standard digital testing, the nail state context of all test nails is preserved across the FAST and END FAST statements.

END BURST — Keywords identifying the last statement in the burst.

Examples

1. The following simple example illustrates how a burst could be programmed to test a single NAND gate in a type 7400 chip. (Note that a burst would normally test all gates in the chip; however, only one is being used to simplify the example). The gate and its truth table are shown in Figure 6-14.

The test program segment would appear as follows:

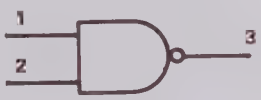
```
ASSIGN LGC  VIHA=4.5 VILA=0.0 VOHA=2.1 VOLA=1.0 LVLA(1-3);

U17:   BURST  IST=1.0U  ITT=5.0U  MAXTIME=40M  [BRANCH L7400;];
       IC(1,2)  OS(3) OH(3);
       IH(2);
       IH(1)  OL(3);
       IL(2)  OH(3);
       END BURST;
```

Meaning: A brief explanation of each statement is given below. For more details on the purpose and format of each command, refer to the individual description of that command.

Prior to the burst, logic voltage levels must be assigned via the ASSIGN statement. The driver high and low levels are +4.5 V and 0 V; the corresponding sensor threshold levels are +2.1 V and +1.0 V. At the start of the burst, all nails are assumed disconnected, ignored and low.

The BURST statement defines the following burst parameters: At each test step, the time between driving the inputs and checking the outputs will be 1 us (IST); the time between test steps will be 5 us (ITT); and the maximum time for executing the burst will be 40 ms (MAXTIME). A fail condition will occur if the 40 ms time-out interval is exceeded. At the end of the burst, a program branch to L7400 will occur if a failure was detected in the burst.



PINS	1	2	3
	L	L	H
	L	H	H
	H	L	H
	H	H	L

Figure 6-14. Single NAND Gate and Truth Table.

The 4 logic test instructions following the BURST statement test all nails for all combinations of inputs to the NAND gate. The 4 statements perform the following functions:

1. IC(1,2) connects the driver outputs to UUT nails 1 and 2; OS(3) enables the sensor circuit on UUT nail 3. Input nails 1 and 2 are driven low (the initial state of all nails at the start of a burst) and output nail 3 is tested for a high state (greater than +2.1 V), as specified. Note that, unless specifically ignored, input nails 1 and 2 are also checked for a low state.

2. IH(2) drives input nail 2 high and all nails are again tested. Note that nail specifications are incremental and, therefore, input nail 1 and output nail 3 remain in their previous state until explicitly changed.

3. IH(1) drives input nail 1 high and OL(3) causes output nail 3 to be checked for a low state.

4. IL(2) drives nail 2 back low and OH(3) indicates that nail 3 is to be tested for a high state.

2.
```
U25: BURST NOPROBE - - - ;
                '
                '
     END BURST;
```

Meaning: The system will <u>not</u> direct the user to probe IC U25, upon failure of the burst.

3.
```
DECLARE BSTRING DIGNAILS(64);
        '
        '
U13: BURST..... INTO DIGNAILS;
        '
        '
     END BURST;
        '
        '
     IF DIGNAILS (29) THEN
     [WRITE 'TEST NAIL 29 FAILED, SKIP U14 TEST';
     BRANCH U15;];
```

Meaning: If a failure is detected at digital test nail 29, set a 1 in bit 29 of bit string variable DIGNAILS. Test variable DIGNAILS. If bit 29 is set to 1, display the message and branch to U15.

4. Similar ICs, such as memory chips with common address and control lines, can be parallel-tested in a single burst. The simple example shown in Figure 6-15 illustrates this procedure for testing 2 RAM chips with common address and control inputs, but separate data lines.

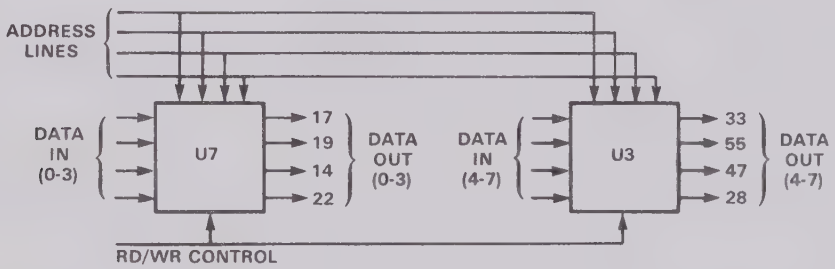


Figure 6-15. Parallel Testing Example.

```
RAM: BURST . . . NAME='U3,U7';
        '
     test statements
     (treating both chips as one)
        '
        '
     END BURST
```

Meaning: Treat the 2 ICs (U3 and U7) as a single chip (labeled RAM in this burst.) NOTE: In the test statements, be sure to account for the outputs of both ICs when testing logic states.

If a failure occurs, the system will automatically compare the connection data in the .IDD file with the named ICs. At the end of the burst, if a failure was detected, the [error-command] is executed, causing the failed IC to be INDICTed (i.e., treated the same as an IC that failed a burst test).

Rules and Considerations

1. A burst may only contain the following type of statements: SET LGC, TEST LGC, GOTO, GOSUB, SUBROUTINE/RETURN/END SUBROUTINE, BREAK and modifiers LOOPIF, EXITIF and OR.

2. The END BURST statement cannot be labeled.

3. An [error-branch] on a BURST statement can contain only one BRANCH statement. If an error is detected, the [error-branch] is executed after the burst is complete.

4. Similar ICs such as RAM chips can be tested in parallel within a single burst, as illustrated in Example 4.

5. The nail-state context is normally defined only up to the end of the burst, unless the burst begins with BURST ACTIVE. After a BURST ACTIVE statement the context remains defined until the next CLEAR LGC or BREAK CLEAR statement, or until the END BURST of a BURST that begins with BURST CLEAR (or simply BURST). The 2272 clears all scanner connections at the end of a context block, so any analog statements associated with a burst (e.g., for testing D/A devices) should be placed after a dummy BURST ACTIVE; END BURST.

6. FAST sections and other burst sections that reference test nails in the SpeedPlus subsystem cannot use logic voltages (refer to ASSIGN LGC) outside of the range -6 to +10 V.

7. In FAST sections the maximum LOOP count is 65535. An EXITIF within the LOOP may not appear within the last 5 statements before the END LOOP statement.

8. A GOTO must not cause a jump into, out of, or between FAST sections.

9. GOSUB and the TEST LGC OM() term are not recognized within FAST sections.

----------BUSTEST/END BUSTEST----------

General

The BUSTEST construct of the 227X test language helps to diagnose static bus failures and to isolate the cause of the failure to one or more defective devices on a bus. Typical causes of bus failures are an output transistor always turned on due to an internal fault in the device, or a faulty select line which leaves the device constantly enabled. In the following description, bus test refers to the initial go/no-go testing of all common bus lines in parallel. Subsequent to the bus test, the BUSBUSTtm diagnostic technique can be used to identify the component causing the bus to fail. While bus test and BUSBUST sections can be written manually, they are more commonly written by the system's Automatic Test Generation (ATG) software, which analyzes the UUT circuit description (.CKT file) and consults system IC libraries to find test procedures required for the various ICs on the bus. Bus tests can be performed using both the standard digital test facilities and all features of the optional SpeedPlus subsystem.

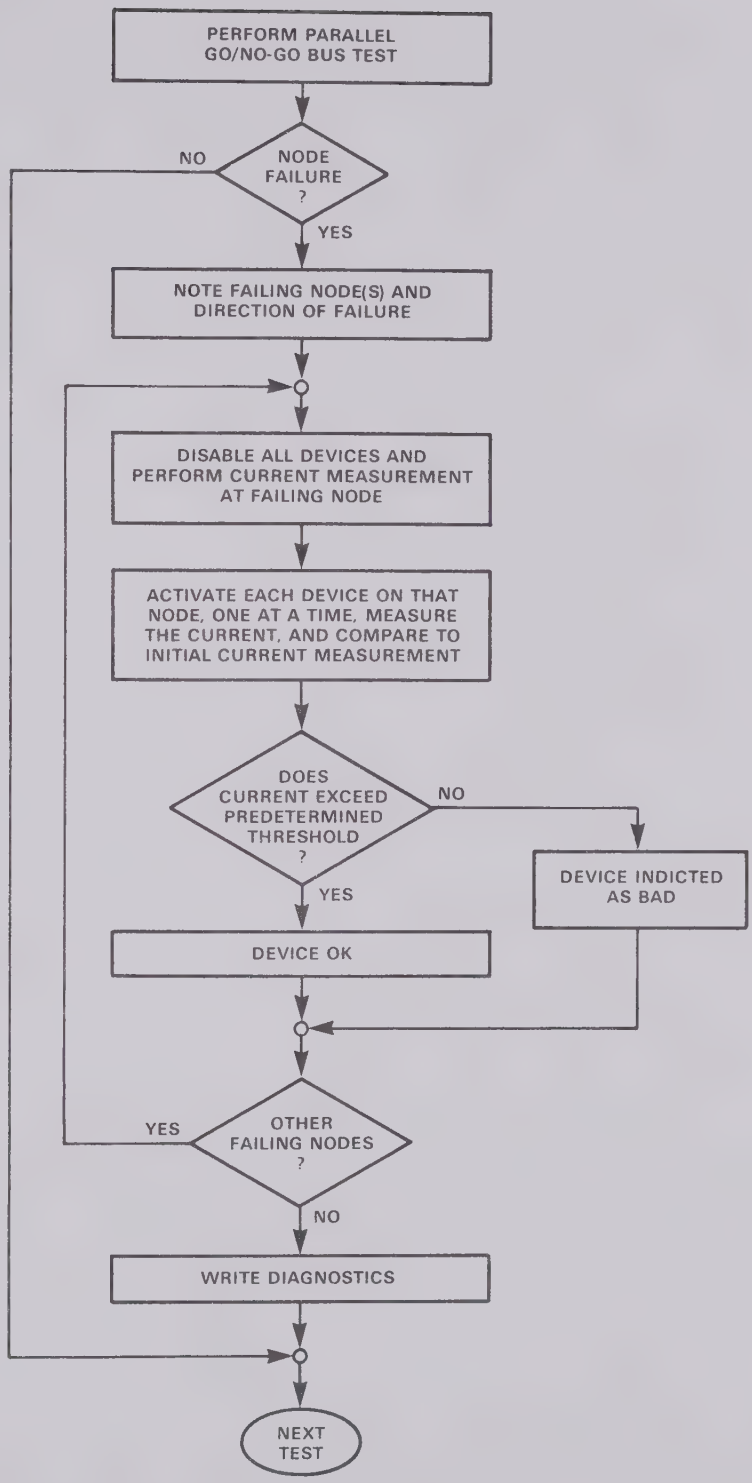

PERFORM PARALLEL GO/NO-GO BUS TEST
NODE FAILURE ?
NO
YES
NOTE FAILING NODE(S) AND DIRECTION OF FAILURE
DISABLE ALL DEVICES AND PERFORM CURRENT MEASUREMENT AT FAILING NODE
ACTIVATE EACH DEVICE ON THAT NODE, ONE AT A TIME, MEASURE THE CURRENT, AND COMPARE TO INITIAL CURRENT MEASUREMENT
DOES CURRENT EXCEED PREDETERMINED THRESHOLD ?
NO
YES
DEVICE INDICTED AS BAD
DEVICE OK
OTHER FAILING NODES ?
YES
NO
WRITE DIAGNOSTICS
NEXT TEST

Figure 6-16. BUSTEST/BUSBUST Procedure.

Figure 6-16 shows the steps taken by a typical BUSTEST block for testing a bus and diagnosing bus failures. The initial go/no-go bus test consists of disabling all devices sharing common bus lines and of using system pull-up and pull-down resistors to verify that the buses are not stuck in one state. If a

failure is detected, all devices on the failing bus are disabled, and the current-sourcing or sinking capability of that bus node is measured. Then, one device (IC) at a time is instructed to drive the node in the direction of the failure (e.g., if the node is stuck low, the IC inputs are driven to force the output low), and the resulting current is measured. If there is no significant increase in current, then the IC is assumed to be bad, since it must have already been driving the node when all devices were disabled.

The following paragraphs describe the two BUSTEST test sections in detail.

A. Parallel Bus Test

In typical test preparation, ATG identifies the common bus for all bused devices, then extracts procedures from the test library to disable all devices connected to the common bus. (These procedures appear in the DISABle block, shown later in the FORMAT section of this description.) During program execution of a bus test, the system disables all devices on the bus and tries to pull each line up and down via the pull-up and pull-down resistors on the digital driver sensor circuits. (Note: Open-emitter and open-collector devices are pulled only in one direction.)

Figure 6-17 shows a simple example of a parallel bus test, along with the test language statements required to implement it. (Refer to subsequent paragraphs describing BUSTEST format for details.) Note that the 2 bus lines (at test nails 7 and 8) are tested in parallel by 2 separate driver/sensor circuits.

```
        ASSIGN LGC  VIHA=4.5  VILA=0,0  VOHA=2.2  VOLA=1.4  LVLA(1-8);

        /*BUSTEST Statement*/
B1:     BUSTEST AT(7-8) IST=IOU;


        IC(3) IH(3); /*Disable IC1*/
        IC(6) IH(6); /*Disable IC2*/

        /*Parallel Bus Test*/

        BUSDATA;
        $PD (7-8);
        OS(7-8) OL(7-8);
        $PN(7-8);
        $OI(7-8);
        $PU(7-8);
        OS(7-8) OH(7-8)
        $PN(7-8);
        $OI(7-8);
```

Note: Refer to Format paragraph for a description of this program segment.

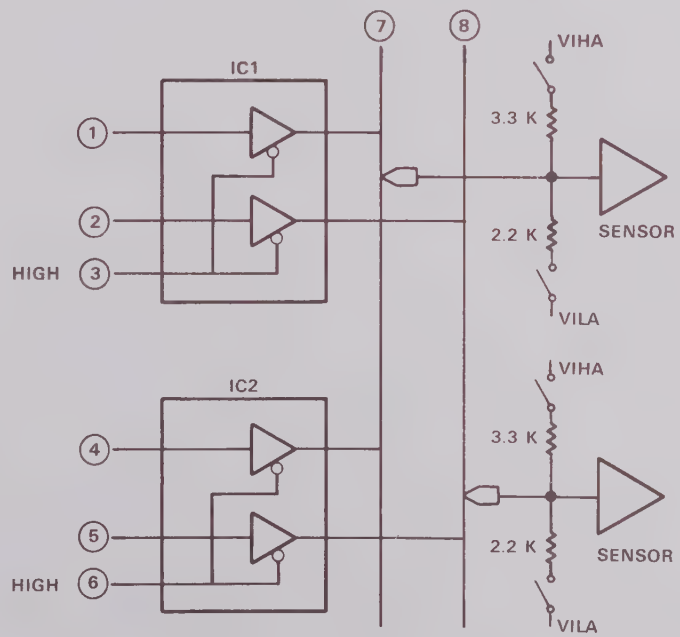


Figure 6-17. Parallel Bus Test.

If this bus test fails, that is, if either node 7 or 8 is stuck high or low, the system flags the failing node(s), notes the direction of the failure (stuck high or stuck low) and proceeds with the following section, which constitutes the BUSBUST component isolation section. Otherwise (if no failure is detected) the bus test is complete and system goes on to the next device test.

B. Bus Disable Measurement (BUSBUST technique)

The first step in the BUSBUST procedure is to disable all devices on the first failing node (line) of the common bus and make an analog measurement of the current flowing into or out of the node. This current measurement then serves as a standard against which to compare subsequent measurements made while each device is turned on individually.

The measurement is made by attaching various system instruments to the node, as shown in Figure 6-18. The 1-ohm resistor, from the Self-Test Module (STM), combined with the DC Voltage Measurement unit form an ammeter circuit. (The system calculates the current by measuring the voltage across a known resistor.) The voltage and current sources are used to source or sink current to the node at a given voltage level. The voltage source will sink or source approximately 160 mA and the current source, 128 mA.

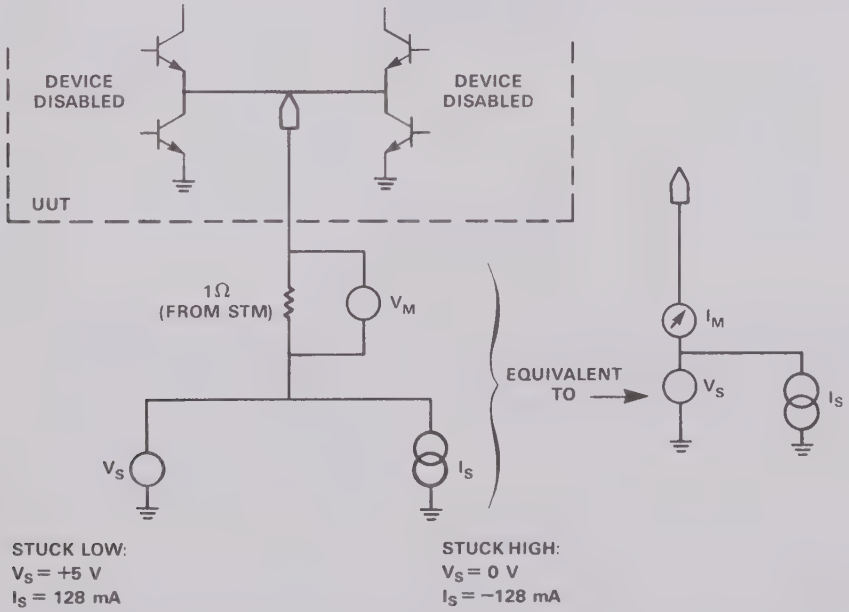


Figure 6-18. Disables Bus Current Measurement.

The value and polarity to which the source is programmed depend on the direction of the failure.

If the pull-up test failed, the bus is stuck low:

1. The voltage source is set to +5 V,
2. The current source is set to +128 mA, and
3. The LFORCE sections (see Format description) are used in subsequent measurements to force each device output low.

If the pull-down test failed, the bus is stuck high:

1. The voltage source is set to 0 V,
2. The current source is set to -128 mA, and
3. The HFORCE sections (see Format description) are used to force each device output high.

C. Force Individual Devices to Failing State

Once the initial current measurement is made at the failing node, each device is individually driven (via its inputs) to force its output to the failing state, and another parametric measurement is made using the same instrument configuration. The procedures used to force each output to the failing state are denoted by the keywords LFORCE or HFORCE in both the library model and the .TPG test program file. These procedures for all bused ICs are typically included in the test library, in the same manner as the disables and inhibits. (Refer to the 227X Test Library Programming manual, Section 6.) HFORCE and LFORCE procedures for any custom IC can easily be included in a library element, as shown in the example of Figure 6-19.

```
.HEAD
        .SIZE 8;
        .INPUT (1=IN1, 2=IN2, 3=CS)
        .TRI (5=O1, 6=O2)
.END HEAD

.DISAB (
.END DISAB;

.INH (
.END INH;

.HFORCE (O1,O2);
  IL(CS) IH(IN1,IN2);
.END HFORCE;

.LFORCE (O1,O2);
  IL(CS,IN1,IN2);
.END LFORCE;

.MAIN
```

Figure 6-19.
HFORCE/LFORCE Procedures
in Test Library.

D. Compare Current Measurements

With one device enabled, a new current measurement is made and compared to the original measurement when all devices were disabled. If the new measurement does not exceed the original measurement by a specified percentage or if the difference does not exceed a specified value, then the device is indicated as a bad device. This current threshold can be specified by the parameters RATIO= or DELTA= in the BUSTEST statement and, if needed, in the individual HFORCE/LFORCE statements. These individual threshold specifications would typically be used when high-current buffers share a common node with low-current (e.g., CMOS) devices. The default threshold value for a good IC corresponds to a 30% increase in measured current, when that IC is turned on.

NOTE

Two DEBUG sub-modes, HBUS and LBUS, can be used to exercise the HFORCE and LFORCE statements and show the user the various BUSBUST current measurements on any bus node. By having available the actual bus current measurements associated with each device, the user can then adjust individual RATIO= or DELTA= parameters, as needed. Refer to the System Operation manual for details on DEBUG sub-modes.

As shown in Figure 6-20, each device on the failed node is turned on, one at a time, the current measured, and the comparison made, to verify the operation of each device on that bus node. This procedure is then repeated for all failing nodes.

The following occur after the failing bus node has been analyzed via the BUSBUST procedures:

1. The failure is flagged (by the letter B) in the data log file.

2. A failed bus test message is generated and the standard diagnostic and probing procedures that apply to burst testing are exercised.

3. If no IC device has been indicted, the failed bus node is still identified by the following typical message:

    N153 -Bad bus node

Format

The programming structure for implementing bus tests consists of a BUSTEST block composed of the following sections:

A. BUSTEST statement

In addition to identifying the beginning of the bus test block, this statement defines various default test parameters such as the common bus nodes to be tested, timing and current threshold for indicting an IC.

B. Disable block

This block contains as many digital statements as required to disable all ICs that drive the bus lines. These statements are executed prior to the initial go/no-go bus test and are repeated prior to a BUSBUST current measurement, when all devices are disabled.

C. BUSDATA block

The digital statements in this block combined with those in the Disable block perform a go/no-go test of the bus (with all inputs disabled). If no bus failures are detected, or if a failure is detected but the failing node is not connected to a hybrid nail or the BUSTEST block contains no Force section, then bus testing ends at the completion of this go/no-go test.

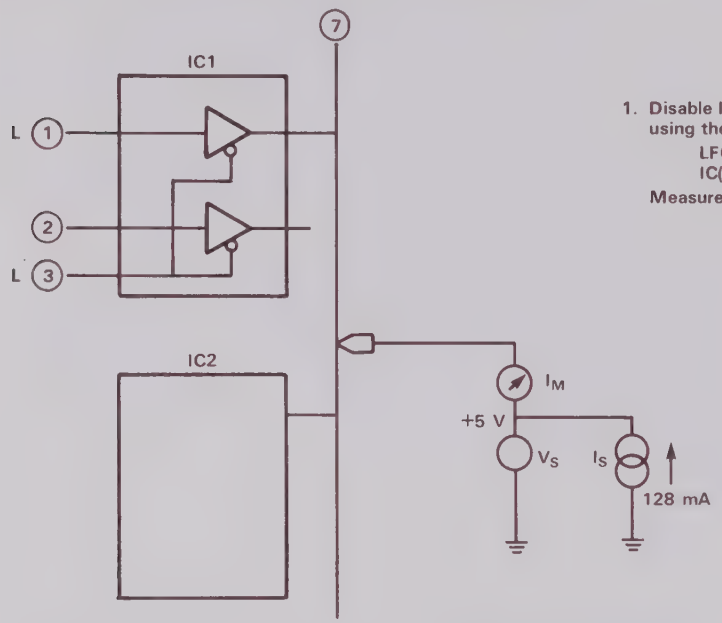
D. Force block

This block is used for diagnosing bus failures. The force statements are not required for the go/no-go bus test but are mandatory for the subsequent

BUSBUST current measurements. Statements within this section specify, for each IC, the device inputs required to force the outputs high and low. Appropriate digital statements from this section are appended to the Disable block when attempting to locate the source of a bus failure.

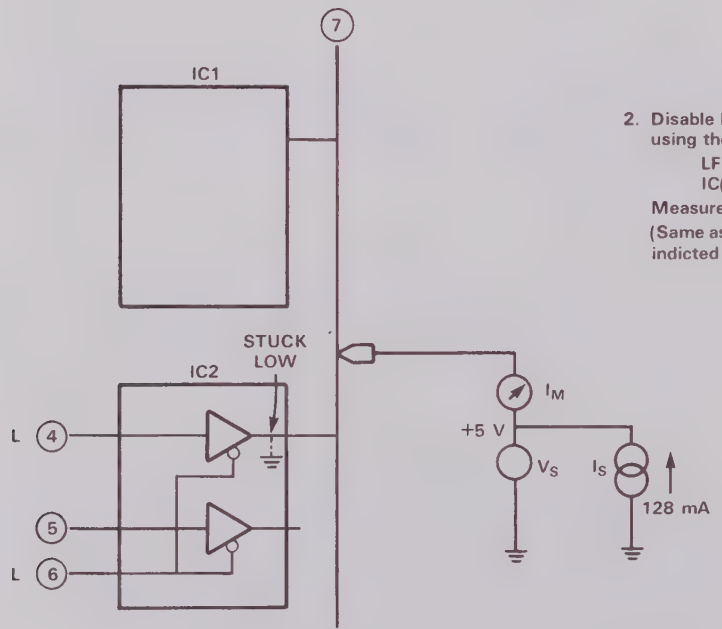
E. END BUSTEST statement

Identifies the end of the bus test block.



1. Disable IC2 and force IC1 output to bus node 7 low, using the following LFORCE statement:
   LFORCE NAME = 'IC1' AT(7);
   IC(1,3) IL(1,3);
   Measured current (IM) = 200 mA

DRIVE IC1 OUTPUT (7) LOW.



2. Disable IC1 and force IC2 output to bus node 7 low, using the following LFORCE statement:
   LFORCE NAME = 'IC2' AT(7);
   IC(4,6) IL(4,6);
   Measured current (IM) = 100 mA
   (Same as with IC1 and IC2 disabled, therefore, IC2 is indicted as bad.)

DRIVE IC2 OUTPUT (7) LOW.

Figure 6-20. Sample test with bus node (7) stuck low, caused by IC2 failure. For this example, assume that the measured current, with both IC1 and IC2 disabled, = 100 mA.

Format

```
label:    BUSTEST AT(pin-list)                                    ┐
          DELTA=value  or  RATIO=value                            │
          NAME='ic-name(s)'                                       │
          IST=value    ITT=value                                  │ BUSTEST
          NOPRINT      NOPROBE                                    │ STATEMENT
          INTO bstring FAIL(n);                                   │
                '                                                 │
                '                                                 ┘
       digital statements required to disable the bus lines;      ┐
       digital statements required to inhibit devices on bus;     │ DISABLE
                '                                                 │ BLOCK
                '                                                 ┘
       BUSDATA;                                                   ┐
       digital statements for go/no-go test;                      │ BUSDATA
                '                                                 │ (PU/PD STATEMENTS)
                '                                                 ┘
       LFORCE                                                     ┐
          AT(pin-list)                                            │
          DELTA=value  or  RATIO=value                            │
          NAME='ic-name(s)'                                       │
          FAIL(n);                                                │
                '                                                 │
                '                                                 │
       digital statement to force specified bus driver outputs low;│
                        '                                         │
                        '                                         │
              (repeat for each bus driver IC)                     │ HFORCE/LFORCE
                        '                                         │ SECTION
                        '                                         │
       HFORCE  {same as LFORCE options};                          │
                        '                                         │
                        '                                         │
       digital statement to force specified bus driver outputs high;│
                        '                                         │
                        '                                         │
              (repeat for each bus driver IC)                     │
                        '                                         │
                        '                                         │
       END BUSTEST  [error-command;];                             ┘
```

where: label — Provides a reference name for the bus test. The label consists of the letter B, indicating a bus test, followed by the first number from the nail-list and terminated by a colon (e.g., B25:).

BUSTEST — Keyword indicating the start of the bus test.

AT(nail-list) — Specifies the bus nails involved in the BUSTEST or LFORCE/HFORCE statement. The AT( ) parameter is required in the BUSTEST statement but is optional in the LFORCE and HFORCE statements. If AT( ) is not specified in a FORCE statement, then the nail-list reverts to nail-list specified in the BUSTEST statement.

When only hybrid test nails are used (e.g., in a 2271 or 2272 system), the nail list format is simply:

AT(nail-list)

However, in 2270 systems containing pure digital and/or pure analog test nails, both a digital and analog test nail must be connected to each bus node to be analyzed

AT(nail-list) (cont)

by the BUSBUST procedure. To specify this double nail the following format is used:

AT(hybrid/analog-nail-list:hybrid/digital-nail-list)

Note: If any hybrid nails are specified, they must be included in both the digital and analog fields. For example,

AT(1-7,17:1-7,53)

where test nails 1-7 are hybrid, 17 is pure digital and 53 is pure analog.

DELTA=value or RATIO=value

Two separate methods of specifying the acceptance level of the measured current required to exonerate a bus driver IC. The value specified by the DELTA parameter indicates the minimum increase in measured current (with the forced IC output) needed to exonerate the IC. Similarly, the value specified by the RATIO parameter indicates the minimum current ratio (with and without the forced IC) sufficient to exonerate the IC.

NAME='ic-name(s)'

May be used in a BUSTEST or LFORCE/HFORCE statement to specify the IC (or ICs) being tested by that statement. Multiple entries are separated by commas (e.g., NAME='U18,U19'). If NAME= is not specified in an LFORCE/HFORCE statement, the label associated with the BUSTEST statement is assumed as default.

IST=value
ITT=value

As in a burst, these 2 values specify the Inter-Strobe Time (IST) and Inter-Test Time (ITT) for the digital test steps contained in the bus test block. (Refer to BURST description for details.)

NOPRINT

Optional parameter used to suppress all diagnostic messages resulting from a failure in this bus test.

NOPROBE

Optional parameter used to inhibit probing the IC specified in the BUSTEST statement.

NOSYNC/SYNC

Unless the keyword NOSYNC is present in the BUSTEST or FORCE section header, the following verification is made:

A. At the beginning of every BUSTEST section which contains FAST sections, the system waits for clock synchronization. If no synchronization is achieved the BUSTEST section exits as FAILED and writes the message "NO SYNCHRONIZATION ACHIEVED" to the MESFILE.

B. At the end of every BUSTEST section which contains FAST sections, the system checks if synchronization was maintained throughout the BUSTEST. If synchronization is lost at any time during the BUSTEST this will be reported in the MESFILE with the message: "SYNCHRONIZATION LOST."

If the NOSYNC is present in the BUSTEST the above checks will not be made.

If either keyword is present in a BUSTEST header, it will apply to all sections of the BUSTEST. If it appears in a FORCE section header, it applies only to that FORCE section.

INTO variable — Optional parameter specifying a bit-string variable into which test nail failures are stored after the bus test is complete. Each bit position in the variable represents a test nail; bit 1 corresponds to nail 1, bit 2 corresponds to nail 2, etc. A 1 indicates a nail failure, a 0 indicates no failure. If the bit-string variable is not long enough to handle the nail-list, a warning message will be printed and the system will attempt to continue with the burst and ignore the missing bit positions.

FAIL(n) — Optional. Fail vector bit n. Set to 1 after the bus test if a failure occurs within the test. If n is a null (i.e., FAIL( )), then no fail bit is set. If FAIL(n) is not specified at all, then fail bit 1 is set upon detecting a failure.

BUSDATA — Statement defining the start of bus data block (go/no-go test).

LFORCE/HFORCE — Keywords used in the force block to indicate whether the specified IC output will be forced high (HFORCE) or low (LFORCE).

END BUSTEST — Keyword identifying the last statement in the bus test block.

[error-command;] — Optional parameter in an END BUSTEST statement. Executed if the bus test fails.

Example

1. The following simple example illustrates the BUSTEST construct for testing the 4-bit bus shown in Figure 6-21.

```
BUSTST1: /*BUSTEST statement*/

B10:    BUSTEST AT(10-13)  ITT=10U  RATIO=1.1
          NOPROBE  FAIL(12);

        /*Disable all bus lines*/

        IC(20-22)  IL(20)  IH(21,22);

        /*Inhibits as needed*/
        /*Go/no-go test**/
```

```
BUSDATA;
    $ PD(10-13); $ ;
      OS(10-13) OL(10-13);
    $ PN(10-13); $ OI(10-13);
    $ PU(10-13); $ ;
      OS(10-13) OH(10-13);
    $ PN(10-13); $ OI(10-13);

    /*Force section*/

LFORCE NAME='U60'; /*Requires Inew/Ibase > 1.3 to pass U60*/
    IC (20,30-33)  IH(20)  IL(30-33); /*Force all U60 outputs low*/

LFORCE NAME='U61' AT(10-11) RATIO=1.5;
     /*Requires Inew/Ibase > 1.5 to pass U61*/
    IC(21,40-41)  IL(21,40-41); /*Force two U61 outputs low*/

LFORCE NAME='U62' AT(12-13) DELTA=26U;
      /*Requires at least 26 uA extra current to pass U62*/
    IC(22,50-51)  IL(22)  IH(50-51); /*Force two U62 outputs low*/

HFORCE NAME='U60';  /*Force all U60 outputs high*/
    IC(20,30-33)  IH(20,30-33);

HFORCE NAME='U61' AT(10-11); /*Force two U61 outputs high*/
    IC(21,40-41)  IL(21)  IH(40-41);

HFORCE NAME='U62' AT(10-11); /*Force two U62 outputs high*/
    IC(22,50-51)  IL(22,50-51);

END BUSTEST [BRANCH BUSERR;];
```

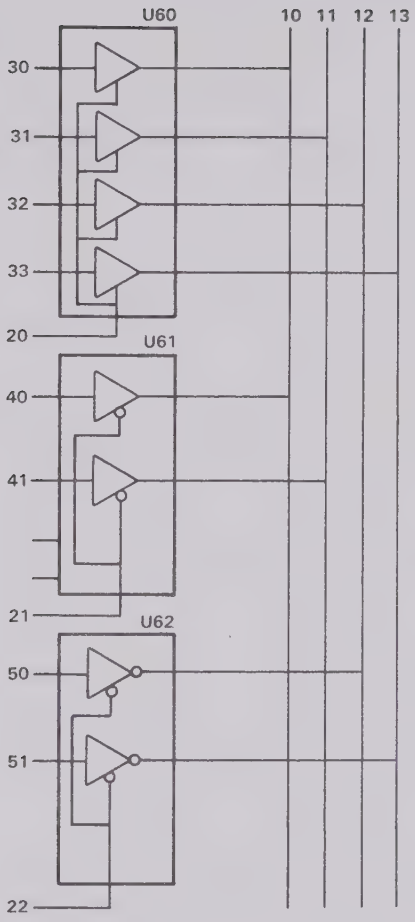


Figure 6-21. Bus Test Example.

Meaning: The BUSTEST statement indicates the start of the bus test block and defines the following parameters. The 4 bus lines being tested are connected to test nails 10-13. The time between digital test steps (ITT) is 10 us and the time between driving the inputs and checking the outputs (IST) assumes the default value of 1 us. A RATIO of 1.3 determines that an IC under test is exonerated if it can increase bus current 30% with all other ICs disabled. This value can be temporarily superseded by a RATIO or DELTA setting in a FORCE statement. NOPROBE states that, if a failure occurs during the test, the test will inhibit probing ICs associated with the bus and will set fail bit 12.

In preparation for a go/no-go test of the 4 bus lines, disable all ICs that drive those lines by forcing nail 20 low and nails 21 and 22 high. Then parallel-test the 4 bus lines by first connecting them all to pull-down resistors (on the driver/sensors) and testing for all low outputs, then connecting them to pull-up resistors and testing for all high outputs.

If any bus nodes are stuck either high or low, the system first executes the statements in the Disable block to disable all devices on the failed node, and makes a current measurement. It then uses the HFORCE and LFORCE statements to turn on one device at a time, measure the resulting current, and determine if that device is bad.

Each LFORCE and HFORCE statement specifies the name of the IC being forced, the subset of bus nails driven by this IC (if applicable) and the current threshold value (DELTA= or RATIO=) if it is different from the one defined in the BUSTEST statement.

Following each LFORCE (or HFORCE) statement are digital statements that activates the IC inputs as required to drive the IC outputs low (or high).

Rules and Considerations

1. The states of the digital driver/sensors at the end of the Disable block are the states which the driver/sensors will assume at the start of all subsequent blocks.

2. Test nails that fail a BUSTEST (or BURST BUS block - see BURST description) will veto the execution of all later bursts that use those nails.

3. If a bus node is not connected to a hybrid test nail or if the associated bus test block has no Force section, that bus node is still tested via the go/no-go test, but no fault isolation is performed in case of failure.

4. The instrument MUX and SCANner system is cleared whenever a BUSTEST is executed.

5. The untranslator (available in DEBUG mode) cannot be used to modify the bus test block in the .TPG file. Use the incremental translator, instead, if changes have to be made.

6. If the UUT contains ROMs or PALs, the HFORCE and LFORCE sections provided by ATG may have to be modified by the user. For example, since ATG has no knowledge of the actual contents of ROMs, it typically leaves all address lines disconnected. To implement the BUSBUST diagnostic procedures, the user must determine what ROM locations should be addressed to force the desired outputs high or low and then modify the HFORCE and LFORCE statements accordingly to generate those address.

7. When a bus node is connected to a terminating resistor on the board, the ATG-generated bus test may have to be slightly modified. This is particularly true in the case of low-valued terminating resistors (e.g., 510 ohms). The modification consists of adjusting the pull-up or pull-down voltage assigned by ATG in the go/no-go bus test, so that the actual voltage measured at the bus node will correspond to the expected value. This is done by changing the ASSIGN LGC statement that specifies the voltage levels for the test.

8. On systems with the SpeedPlus option, the BUSTEST construct accepts the BURST parameters used with SpeedPlus tests. The BUSDATA, LFORCE, and disable blocks can contain FAST; . . . END FAST; sections. The most recently executed CLOCK section defines timing within these FAST sections.

----------CLEAR CLOCK (SpeedPlus Only)----------

General

On systems with the SpeedPlus option, CLEAR CLOCK disconnects the clock from the UUT and stops all clock signals. (NOTE: The clock is turned off automatically at the end of the test program and at the end of each BURST if CLOCK ON is not specified.)

Examples:

```
CLK1: SET CLOCK AT(1) F=1MEG ON;
U1  : BURST;
      END BURST;
CLK2: CLEAR CLOCK
```

Meaning: The clock drive signals remain on and connected to the UUT from the SET CLOCK statement until CLEAR CLOCK is executed.

----------CLEAR LGC----------

General

This command is used to reset the driver/sensor system. Resetting the driver/sensor system clears the driver/sensor system and places all nails in the disconnected (ID), ignored (II and OI) and low (IL and OL) states.

Format

```
CLEAR LGC;
```

Rules and Considerations

1. No additional parameters may be specified in a CLEAR LGC statement.

2. CLEAR LGC may only be used outside a burst.

----------DECLARE GROUP----------

This belongs to the DECLARE statement, used for defining groups of digital nails. Refer to the DECLARE GROUP description earlier in this section, para 6.2.

----------EXITIF----------

General

This command is used in conjunction with LOOP/END LOOP to specify an additional exit condition. The loop will terminate whenever the EXITIF condition is satisfied (or when the maximum count has been exceeded).

Format

```
    TEST LGC  nail-spec   EXITIF PASSES/FAILS;
or
    EXITIF PASSES/FAILS   TEST LGC nail-spec;
```

where: EXITIF — Optional keyword used in conjunction with a test logic statement to provide a conditional exit from a loop. Loop terminates following the END LOOP statement after the EXITIF condition is reached.

PASSES/FAILS — Optional keywords used with EXITIF to indicate if a "test passes" or a "test fails" satisfies the EXITIF condition. PASSES is the default condition.

Examples

1.
```
LOOP IH(10);
EXITIF IL(10) OL(11-13)  END LOOP;
```

   Meaning: Keep pulsing input nail 10 until output nails 11, 12, and 13 are all low.

2. Note that in Example 1, a LOOPIF FAILS would have accomplished the same results as the EXITIF. For example:

```
LOOP IH(10);
LOOPIF FAILS IL(10) OL(11-13) END LOOP;
```

   Meaning: Keep pulsing input nail 10 as long as output nails 11, 12 and 13 are not all low. When these 3 nails are all low, terminate the loop.

   In general, if the loop or exit condition can be expressed in a single statement, equivalent expressions can be written using either LOOPIF or EXITIF with the appropriate PASSES/FAILS keyword. The distinction between LOOPIF and EXITIF becomes apparent, however, when OR conditions are specified (see Example 3 in the OR description).

Rules and Considerations

1. EXITIF can only be used within a burst as part of a LOOP statement.

2. Only one EXITIF (or LOOPIF) can be used within a loop.

3. EXITIF is not allowed with a SET LGC statement since a SET LGC statement always PASSES.

4. EXITIF can appear anywhere in a statement except between the command keywords and the nail specification.

5. Once an exit condition is detected, execution continues to the END LOOP statement and then terminates.

6. The only condition that affects EXITIF is the result of the logic test in the EXITIF statement itself. A PASS or FAIL condition anywhere else in the loop has no effect on EXITIF.

----------FLAGFAIL (SpeedPlus Only)----------

General

Within a FAST section of a BURST, FLAGFAIL specifies that the local fail flag should be set if the test step containing the FLAGFAIL term fails. (Standard FAIL bits are not accessible within a FAST section.) This flag determines the action taken by the statements

```
GOTO label FLAG PASSES
GOTO label FLAG FAILS
```

FLAGFAIL must precede either of these statements by at least 5 program statements. The flag is cleared only by a subsequent GOTO within the FAST section. For further information, refer to GOTO.

Example

```
1.  IH(4001) OS(4002) OL(4002) FLAGFAIL;
        ¦
        ¦ <at least 5 test statements>
        ¦
    GOTO U3FAILURE FLAG FAILS;
```

Rules and Considerations

1. FLAGFAIL should be followed by at least one GOTO label FLAG PASSES/FAILS in the same burst to ensure that the flag is cleared.

----------FLOOP/END FLOOP (SpeedPlus Only)----------

General

To allow precise control of loop timing, a slightly altered loop control structure is used within FAST sections, where LOOP and END LOOP are not recognized. Included within the initial FLOOP statement is an (optional) initialization statement that executes just once. If this statement is omitted, the previous test statement (i.e., the nail context), is repeated. A FLOOP fails if at any time a test statement within the loop fails.

Format

```
        initialization-terms    FLOOP=count;
```

where:

initialization-terms	Drive and sense terms that execute prior to entering the loop. These execute just once.
FLOOP=	keyword that identifies beginning of FAST loop.
count	an integer (no greater than 32767) that specifies how many passes through the loop shall be executed.

Example

```
1.  IH(4002) OL(4008) FLOOP=10;  /*IH(4002) OL(4008) happens just once*/
        IH(4010);                /*IH(4010); IL(4010) happens 10 times*/
        IL(4010)
    END FLOOP;                   /*no semi-colon before END FLOOP!*/
```

Meaning: Execute the initialization terms IH(4002) OL(4008) just once; then execute the loop consisting of IH(4010); IL(4010) 10 times.

----------GOSUB----------

General

This command is used in a burst to transfer control to a sequence of statements (subroutine) and later return to the main flow. GOSUB may be either unconditional or conditional upon the results of the previous statement.

Unlike the CALL (subroutine) statement, GOSUB cannot contain an argument list. All variables in the subroutine must be defined within the burst block or in a higher level block. GOSUB is not valid within a FAST section of a burst.

Format

GOSUB name PASSES/FAILS/null;

where:	GOSUB	Command keyword. Must appear first in the statement. Note that GOSUB is a single word, with no space between GO and SUB.
	name	Name of the subroutine.
	PASSES/FAILS/null	Optional. PASSES causes the transfer to occur if the previous statement (normally a TEST LGC statement) was successful. FAILS causes a transfer if the statement failed. Null causes an unconditional transfer.

Example

```
1.  GOSUB TEST1 PASSES;
```

Meaning: If the previous statement PASSed, execute the previously-defined subroutine TEST1, then return to the statement following GOSUB.

Rules and Considerations

1. There is no spacing between GO and SUB.

2. GOSUB may not be used within a FAST section.

3. GOSUB can only be executed to a single level, that is, a GOSUB command is not allowed within a subroutine that is called by another GOSUB.

4. A conditional GOSUB statement should always be preceded by a TEST LGC statement, which can pass or fail. A SET LGC with a nail specification will always pass. A SET LGC with a terminating resistor specification has no effect on the current passes/fails status of the system.

----------GOTO----------

General

Used only in a burst to cause a program transfer to the specified label (cf. BRANCH in para 6.2). The GOTO may either be unconditional or it may depend on the pass/fail condition of the previous statement. GOTOs that branch into, out of, or between FAST sections are not permitted. The GOTO structure used in FAST sections differs slightly from that used in standard BURST sections.

Format

GOTO label PASSES/FAILS/null;

where:

GOTO — Command keyword. Note that GOTO is a single word, with no spacing between GO and TO.

label — Label on the statement to which the program will transfer.

PASSES/FAILS/
FLAG FAILS/FLAG PASSES
PHASELOCK/
NOPHASELOCK/
null

Optional. In standard (non-FAST) sections, PASSES causes the transfer to occur if the previous statement (normally a TEST LGC statement) was successful. FAILS causes a transfer if the previous statement failed. Null causes an unconditional transfer.

On systems with the SpeedPlus option, a FAST section of a burst must, instead, specify a branch on the special FLAGFAIL flag. If FLAGFAIL has been set by the failure of a previous test step, the statement GOTO label FLAG FAILS will execute; otherwise the statement GOTO label FLAG PASSES executes. Both statements clear FLAGFAIL, whether or not the specified branch is taken. FLAG FAILS and FLAG PASSES must appear 5 or more test steps after the FLAGFAIL term appears in the FAST section.

Upon execution of a SET CLOCK statement, the SpeedPlus clock module must either stabilize at the programmed frequency or synchronize itself to an external signal. When the proper frequency or synchronization is reached, the flag PHASELOCK is set; if synchronization is lost, it is cleared. The state of this flag can be tested in a FAST section. GOTO label PHASELOCK executes if the flag is true (set); GOTO label NOPHASELOCK executes if it is false. Neither statement affects the state of the flag.

Example

```
1.  TEST LGC . . . . . ;
    GOTO TEST2 PASSES;
    GOTO NOCLOCK NOPHASELOCK;  /* VALID ONLY IN FAST SECTION */
         '
         '
    TEST2:  SET LGC . . . . . ;
    NOCLOCK:  BREAK;
```

Meaning: A branch to TEST2 will be taken if the TEST LGC statement is successful, i.e., passes. On a system with the SpeedPlus hardware, the burst will be aborted if the clock module has not achieved synchronization by the time the GOTO NOCLOCK statement occurs.

Rules and Considerations

1. There is no spacing between keywords GO and TO.

2. GOTO can only be used within a burst.

3. A conditional GOTO statement should always be preceded by a TEST LGC statement that can pass or fail. For example, the GOTO TEST1 in the following example would never be taken since a SET LGC statement always passes.

```
SET LGC OH(1);
GOTO TEST1 FAILS;
```

An exception to this is a SET LGC statement specifying a terminating resistor (PU, PB, etc). This type of statement has no effect on the current passes/fails status. A conditional GOTO statement following this type of SET LGC would use the passes/fails status of the previous logic test, making the program flow difficult to understand.

4. In a FAST section a GOTO command takes one test step time and executes a test vector identical to the previous statement.

5. Every use of FLAGFAIL should be followed by a GOTO label FLAG PASSES/FAILS

to clear the flag. This flag can only be cleared in this way; it is not cleared between BURSTs.

6. Statements branched around by a GOTO statement are nevertheless displayed in Debug mode by the Digital Display; the displayed pin states for these statements are spurious and should be disregarded.

----------LOOP/END LOOP----------

General

This statement provides the basic loop function within a burst. The sequence of statements to be repeated is bracketed by the keyword LOOP in the first statement and END LOOP in the last statement. A count may be specified indicating the maximum number of times that loop will be executed. Other exit conditions such as LOOPIF and EXITIF are allowed as described in subsequent sections. After the END LOOP statement is executed, the loop will either be repeated or terminated, depending on whether the loop count has expired or other exit conditions have been met. The fail state at the end of the loop is determined solely by the last pass through the loop; LOOP constructs therefore should not be used where a failure might occur only on some intermediate pass, before termination of the loop. (Note: FAST sections must use the keywords FLOOP and END FLOOP to implement this function.)

A LOOP block may not include the statements FAST or END FAST.

Format

```
SET/TEST LGC nail-spec    LOOP=count;
                          '
                          '
SET/TEST LGC nail-spec    END LOOP;
```

where: LOOP — Command keyword that defines the start of a loop. Can either precede or follow the logic command and nail specification, but cannot appear between the two.

=count — Optional. May be used to specify the maximum number of loop iterations. If no count is specified, looping continues until other exit conditions are satisfied or until the burst times-out (as specified by MAXTIME in the BURST statement). Count must be an integer no greater than 32,767.

END LOOP — Command keywords that define the end of the loop and must appear in the last statement in the loop. May precede or follow the logic command and nail specification, but cannot appear between the two.

Example

1.
```
LOOP=4  IH(10);
IL(10)  END LOOP;
```

Meaning: Generate a series of 4 pulses on nail 10.

Rules and Considerations

1. LOOP/END LOOP can only be used in a SET LGC or TEST LGC statement, within a burst.

2. LOOP and END LOOP may precede or follow the logic command and nail specification, but may not appear between the two. For example, LOOP SET LGC IH(1) and SET LGC IH(1) LOOP are both permissible, but SET LGC LOOP IH(1) is not.

3. For efficient use of the driver/sensor memory, include the LOOP/END LOOP specifications as part of a logic statement. Otherwise, 2 memory locations will be used rather than just one, and the logic test will be repeated in both test steps.

4. There must be a separator (i.e., space, tab, etc.) between the keyword END and LOOP. Also, reversing the order of the two words (LOOP END) is not permitted.

5. A loop cannot be used to address nails incrementally by incrementing the group bit string that represents the nails.

6. Keywords LOOP and END LOOP cannot both be included in the same statement.

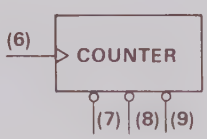
NOTE

As described in para 6.4.3, logic nail specifications are incremental in compile-time order (i.e., the order of appearance in the test program listing) and not necessarily in run-time order (i.e., the order of execution). This distinction is especially significant in a test program loop (or branch), as illustrated in the following example.

Assume that a 3-stage counter is being initialized to a reset state by the following program loop:

```
LABEL1: LOOP=20 IH(6);
LABEL2: IL(6);
LABEL3: END LOOP EXITIF OH(7,8,9);
```

Statement	Test Nail States (6-9)
LABEL1:	H L L L
LABEL2:	L L L L
LABEL3:	L H H H



where L = Low
H = High

Note that, even though the output nails (7-9) will change as the counter is pulsed, the expected states of these outputs will remain L L L during statements LABEL1: and LABEL2:.

The loop will only terminate when nails 7-9 are all high, or after 20 iterations; whichever comes first. However, apparent test failures will develop whenever the 3 outputs are not all low during statements LABEL1: and LABEL2:. These apparent failures may be avoided by preceding the loop with an Output Ignore (OI) nail specification:

```
        OI(7,8,9);
LABEL1: LOOP=20 IH(6);
LABEL2: IL(6);
LABEL3: END LOOP EXITIF OS(7,8,9) OH(7,8,9);
```

or by using SET LGC ($) rather than TEST LGC commands in statements LABEL1: and LABEL2:

```
LABEL1:     LOOP=20 $ IH(6);
LABEL2:     $ IL(6);
LABEL3:     END LOOP EXITIF OH(7,8,9);
```

If either of these approaches is taken, output nails 7-9 will not be tested during statements LABEL1: and LABEL2:, thus no failures will be detected.

As shown in Figure 6-22, the logic commands associated with the LOOP statement and the END LOOP statement are repeated each time through the loop.

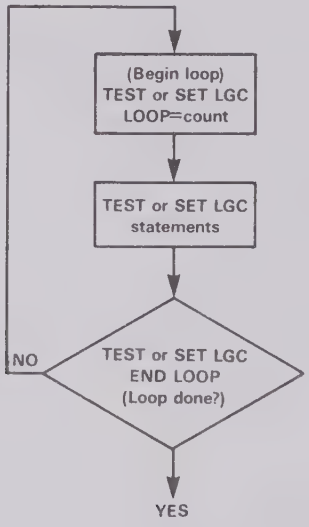


Figure 6-22. Typical Loop Flow Chart.

----------LOOPIF----------

General

LOOPIF must be used in conjunction with LOOP/END LOOP in a burst to specify an additional loop condition, the loop will continue to repeat as long as the LOOPIF statement is satisfied (or the maximum count specified for the loop has not been exceeded). When the LOOPIF condition is no longer satisfied, the loop terminates after executing the test associated with the END LOOP statement.

Format

```
TEST LGC nail-spec   LOOPIF PASSES/FAILS;

        or

LOOPIF PASSES/FAILS    TEST LGC nail-spec;
```

where: LOOPIF Optional keyword used in conjunction with a test logic statement to specify a looping condition. Looping continues as long as the LOOPIF condition is satisfied (and

the maximum loop count has not been reached). LOOPIF can appear anywhere in the statement except between TEST LGC and nail-spec.

PASSES/FAILS Optional keywords used with LOOPIF to indicate if a "test passes" or a "test fails" satisfies the LOOPIF condition. PASSES is the default condition.

Examples

1. LOOP $ IH(10);
   LOOPIF PASSES IL(10) OL(13) END LOOP;

   Meaning: Keep pulsing input nail 10 as long as output nail 13 is low. If nail 13 never goes high, looping will continue until the burst times-out (indicated by MAXTIME in the BURST statement). To prevent this, a maximum loop count could be specified as follows:

2. LOOP=20 $ IH(10);
   LOOPIF IL(10) OL(13) END LOOP;

   Meaning: This will guarantee an exit after 20 iterations even if nail 13 never goes high. Note that PASSES, being the default condition, need not be specified.

Rules and Considerations

1. LOOPIF can only be used as part of a LOOP statement within a burst.

2. LOOPIF is not allowed with a SET LGC statement, since a SET LGC always PASSES.

3. Only one LOOPIF (or EXITIF) specification can be used within a loop.

4. Additional loop conditions can be specified by an OR modifier (see OR description).

5. LOOPIF can appear anywhere in the statement except between the command keywords and the nail-spec.

6. Once the loop condition is no longer satisfied, the loop terminates after executing the END LOOP statement.

7. The only condition that affects LOOPIF is the result of the logic test in the LOOPIF statement itself. A PASS or FAIL condition anywhere else in the loop has no effect on LOOPIF.

----------NAILBIND----------

This belongs to the DECLARE statement and is defined in para 6.2. It is used only on systems with the optional SpeedPlus subsystem to allow testing via the special SpeedPlus clock and trigger nails.

----------OR----------

General

Used in conjunction with LOOPIF and EXITIF to specify alternate conditions for terminating or repeating the loop.

Format

```
TEST LGC nail-spec   OR   PASSES/FAILS;

                or

OR  PASSES/FAILS  TEST LGC nail-spec;
```

where: OR — Optional keyword used in a test logic loop. Appears in a separate statement following a LOOPIF or EXITIF specification to provide additional loop or exit condition(s).

PASSES/FAILS — Optional keyword used with OR to indicate whether the "passes" or "fails" status of the OR will cause the specified looping or exiting condition.

Examples

1.
```
LOOP $ IH(10);
EXITIF IL(10) OL(11) OH(12,13);
OR  OL(12) OH(11,13);
OR  OH(11-13)  END LOOP;
```

Meaning: Keep pulsing input nail 10 until output nails 13, 12 and 11 reach any one of 3 specified states (011, 101 or 111)

2.
```
LOOP $ IH(10);
LOOPIF IL(10) OH(11);
OR  OL(11)  END LOOP;
```

Meaning: Since mid-state is not directly available from the hardware during a burst, this loop provides an indirect means of detecting it. (Note: Mid-state is defined as a level between the high and low sensor thresholds.) Nail 10 will be pulsed until nail 11 is in mid-state.

3. The following examples A and B illustrate the difference between a LOOPIF FAILS and EXITIF PASSES statement operating on the same set of ORed conditions.

A.
```
EXITIF OL(11) OH(12);
OR OH(13);
```

B.
```
LOOPIF FAILS OL(11) OH(12);
OR OH(13);
```

As shown in the following truth table, the 2 examples will cause the loop to terminate under different conditions.

Nails			Example A	Example B
11	12	13		
0	0	0	Loop	Loop
0	0	1	Exit	Loop
0	1	0	Exit	Loop
0	1	1	Exit	Exit
1	0	0	Loop	Loop
1	0	1	Exit	Loop
1	1	0	Loop	Loop
1	1	1	Exit	Loop

Rules and Considerations

1. OR can only be used within a loop and may only follow a LOOPIF, EXITIF or another OR statement.

2. OR is not allowed with SET LGC statements since a SET LGC statement always PASSES.

----------RETURN----------

Refer to RETURN description in para 6.2 - General Utility Statements.

----------SET CLOCK (SpeedPlus Only)----------

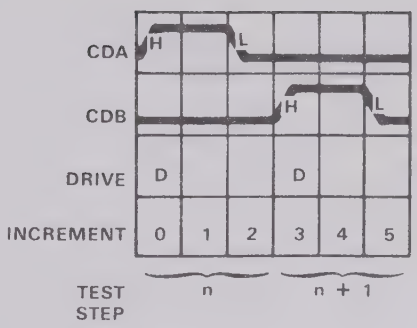
General

On systems equipped with the optional SpeedPlus subsystem, the SET CLOCK statement is used to program the subsystem's clock and timing module. This module can supply clocking waveforms to the UUT at any point in a test (including analog and non-FAST digital test sections), and it determines the timing of FAST sections within bursts or bust tests.

SET CLOCK sets up clock module connections to particular test fixture nails. It also defines the clocking waveforms, frequencies, synchronization, and a number of timing parameters used in FAST sections of bursts or BUSTESTs. These connections and parameters remain in effect until another SET CLOCK or CLEAR CLOCK is executed. The ON parameter causes the clock module to start applying clock signals to the UUT as soon as SET CLOCK executes, so that, for example, dynamic RAMs can retain data across several bursts. Otherwise, clock nails are driven only during bursts; at all other times they are disconnected from the clock driver circuits.

All timing is defined with reference to a matrix of timing vectors. This matrix describes a basic timing cycle that is divided into an arbitrary number of timing increments. At each increment a timing vector represents all of the clock states (high or low) and timing events (such as driver strobes) that occur in that increment. In most cases each timing cycle corresponds to one test step in a FAST section of a BURST or BUSTEST. However, if more than 1 drive strobe is defined in the timing matrix, each drive strobe causes the SpeedPlus subsystem to execute a new test step. Thus, several test steps can

occur in the course of 1 timing cycle. In the example at right, 2 drive strobes are issued in the course of 1 timing cycle, meaning that each timing cycle normally carries out 2 test steps (unless triggering is used to delay some steps). The number of increments in a timing cycle (NINC) must always supplied in the SET CLOCK statement, along with some other term (F, PERIOD, or TINC) that defines the cycle time for the matrix.



Four separate clock signals, designated CDA, CDB, CDC, and CDD can be supplied to the test fixture, and their exact frequencies can be synchronized to a signal on the UUT defined by the parameter CSYNC. Driver/sensor and test step timing in subsequent FAST sections are defined by the parameters DST and F (or PERIOD); additional terms in DST define timing for the triggering mechanism. Logic voltage levels are set by the parameters VCDL, VCDH, and VCST. Note that another important logic level term, VTT, the trigger voltage threshold, is supplied in either the ASSIGN LGC or BURST statement, not in SET CLOCK.

The timing matrix is programmed by simply listing the timing increments in which timing events are to occur. Each event should be regarded as occurring at the very begining of the increment in which it is programmed.

At least once during each timing cycle, the states of the SpeedPlus trigger nails are sampled and stored. Trigger sampling and clock driving occur continuously. Drive and sense events (and the associated test steps), however, are suspended whenever a test step specifies a trigger condition different from the trigger nail states sampled on the most recent T event; they do not resume until the requred trigger states are sampled and an ED event occurs. Note that this provides a mechanism for synchronizing a pattern of test steps to a timing cycle. If the timing cycle contains just one ED event, a test step containing, for example TH(VCC), will wait until the point in the timing cycle where ED occurs. It will, of course, always execute, so long as power (VCC) is applied to the board.

For clock transitions (Ln, Hn) and driver strobes (D), the times specified in the SET CLOCK statement refer to the instant at which the drivers start slewing to their programmed states. However, clock nails and driver nails generally slew at different rates, and therefore they may reach their programmed states at slightly different times. The slew rate of a clock driver is specified by the keywords HISLEW and LOSLEW. Into a typical clock nail, the HISLEW rate is about 250 V/us; the LOSLEW rate is about 125 V/us. A test nail slews at about 70 V/us. Hence, if a typical clock nail and a typical test nail are programmed to change states in the same timing increment, the clock nail will reach the new logic state slightly ahead of the test nail. Except at the highest test rates, the resulting skew is not significant. Potentially more important is the skew at a test nail driver connected to a low-impedance (back-driven) node. When a driver must supply a large amount of current, it slews at a slower rate, and can lag other drivers by as much 200 ns (worst case) for back-driven TTL logic. Consult the Technical Description for your system for the applicable driver slew rates and skews.

Format

```
SET CLOCK
    AT (CDA=nail-list : CDB=nail-list : CDC=nail-list : CDD=nail-list :
        CSYNC=sync-nail)
    F=freq
    PERIOD=period
    TINC=incrtime
    NINC=count
    SIGNAL (CDA=transition-list : CDB=transition-list :
    CDC=transition-list :
            CDD=transition-list : CSYNC=transition-list)
    DST (D/S-timing-list)
    VCDL=volts
    VCDH=volts
    VCST=volts
    HISLEW ¦ LOSLEW
    ON ¦ OFF
    ;
```

where: nail-list — Specifies nails to be driven by clock signals CDA to CDD. The terms CDA= to CDD= may appear in any order, separated by colons (:). They may be omitted (as in, for example SET SCAN AT), in which case the default values of the 4 fields are used. I.e.,

```
AT (1 : 5 : 9)
```

is equivalent to

```
AT (CDA=1 : CDB=5 : CDC=9)
```

From 0 to 4 clock drive signals may be specified. The allowed clock nail ranges for all clock drivers are

```
2271:  1 to 7; 9 to 15
2272:  1 to 16
```

CSYNC=sync-nail — Specifies a nail to be used as input to the clock module from the UUT for clock synchronization. Typically this nail connects to an on-board clock or oscillator line.

To guarantee synchronization, TINC must be in the range of 25 ns to 25.6 us, and the specified (or calculated) frequency should be within ±10% of the actual DUT signal frequency. This frequency range in which clock synchronization will take place is called lock range.

F=freq — The frequency at which the timing matrix is read out. Clock and drive/sense frequencies default to this value, but they may also be multiples of it if more than one drive/sense pair or more than one clock cycle is defined in the timing matrix (i.e., in "transition-list" or in "D/S-timing-list"). Equal to 1/prd. May not be used if PERIOD=prd is used.

PERIOD=prd — The basic period in which the timing matrix is read out. Equal to 1/freq. May not be used if F=freq is used. Range: 50 ns to 51.2 us.

TINC=incrtime — Defines the length in seconds of each increment of the timing matrix. incrtime * count = prd = 1/freq. May not be used if both "count" and either "prd" or "freq" are already defined.
Default: PERIOD/NINC. Range: 25 ns to 51.2 us.

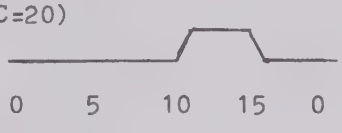
NINC=count — Defines the number of time increments in the timing matrix. count * incrtime = prd = 1/freq. Must be present in every timing definition, and must agree with the value implied by the combination of F or PERIOD with TINC, if these are present. Normally, NINC will be used with only TINC, F, or PERIOD to define the timing matrix. Range: 1 to 256.

SIGNAL — Contains the transition lists that define the clock signals.

CDx=transition-list

Each clock drive signal used can be specified by a transition list. A transition list contains up to 256 transitions, separated by commas. A transition consists of the letter H (for low-to-high) or L (for high-to-low) followed by the increment number in which the level transition takes place. The default transitions are L0 and H< NINC/2> (H< (NINC-1)/2 > for odd NINC); in other words, the default waveform is a square wave with a low-going edge at the first increment and a high-going edge half-way through the timing cycle.

Example: (with NINC=20)
H10,L15



DST(D/S-timing-list)

Used to define the trigger, drive, and sense portions of the timing matrix. A D/S-timing-list is a list of timing control events named with the timing increment in which each event is to occur. The events are separated by commas. Where n represents the increment number, the 4 kinds of events are:

Dn Drive strobe time. Causes the drivers to start slewing to their progammed logic states for the current test step. May not coincide with a sense time, S. Note that the exact time at which the UUT input actually reaches a new logic state depends on the effective slew rate at the test nail and on the driven device's input logic levels.

Default: Every transition in the CSYNC parameter, if specified, otherwise each transition of the lowest lettered clock drive signal parameter present.

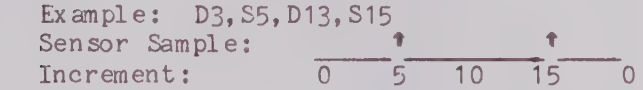
Example: D3,S5,D13,S15

The drivers slew to their new states at increments 3 and 13.

Sn Sense strobe time. Causes the current states of the sensors to be sampled and compared to the expected logic values.

Default: The sense strobe is placed 50 to 200 ns before the next drive strobe (the value used is hardware-dependent and is not set by TINC or NINC).

Example: D3,S5,D13,S15



Tn Trigger sample time. Causes the current states of the trigger nails to be sampled and stored. The stored states remain valid until the next T occurs. Whenever a test step specifies a trigger condition, the specified trigger nail states (TH, TL) are compared to the currently stored states before the step executes.

Like the clock function, trigger sampling continues at all times while the clock module is active (throughout a burst or for as long as the ON parameter is valid). Trigger sampling is not suspended when a test step is suspended

Normally, a T event precedes DS and SS for each test step, and a test step specifying a trigger condition executes if the trigger condition is met immediately before the drive (DS) time. However, if a test step specifying trigger conditions falls on a drive/sense pair that has no T event associated with it, then the trigger nails are not sampled at the beginning of this test step. Instead, the specified trigger states are compared to the trigger nail states that were sampled on the most recent T event.

Default: If no T is specified, the sample trigger strobe is placed 60 to 200 ns before each drive strobe (the value used is hardware-dependent and is not set by TINC or NINC).

EDn Enable Drive time. Whenever a test statement with a trigger condition is encountered, further drive and sense strobes are inhibited and no test steps are executed at least until the next ED occurs. If the most recently sampled trigger nail states are the same as the trigger states specified by the current

(or suspended) test step, testing resumes at an ED. If a match is not present, testing is suspended until both the trigger condition is met at some T (trigger sample) time and a subsequent ED occurs.

Default: If no ED is given, every drive strobe (D) in the D/S-timing-list. That is, if no EDs are present in the list, every D becomes an ED. If T is specified, the default ED is the next D after T. Otherwise, no default is used. At least 1 ED is always in effect.

Example: Consider 3 ways of writing a D/S-timing-list with 1 trigger sample and 2 drive-sense pairs, or cycles.

```
1:  SET CLOCK . . .  /* CLOCK 1 */
        DST(T0,D1,S2,ED3,S4) . . .;

2:  SET CLOCK . . .  /* CLOCK 2 */
        DST(T0,ED1,S2,D3,S4) . . .;

3:  SET CLOCK . . .  /* CLOCK 3 */
        DST(ED1,S2,T3,D4,S5) . . .;
```

Using these different DSTs, the test statements below execute in different ways, depending on how the DST parameter is written. Assume that the first test statement occurs on the first drive-sense pair in the D/S-timing-list.

```
TL(1) TH(2) IH(4001) OH(4004);
            IL(4001) OL(4004);
```

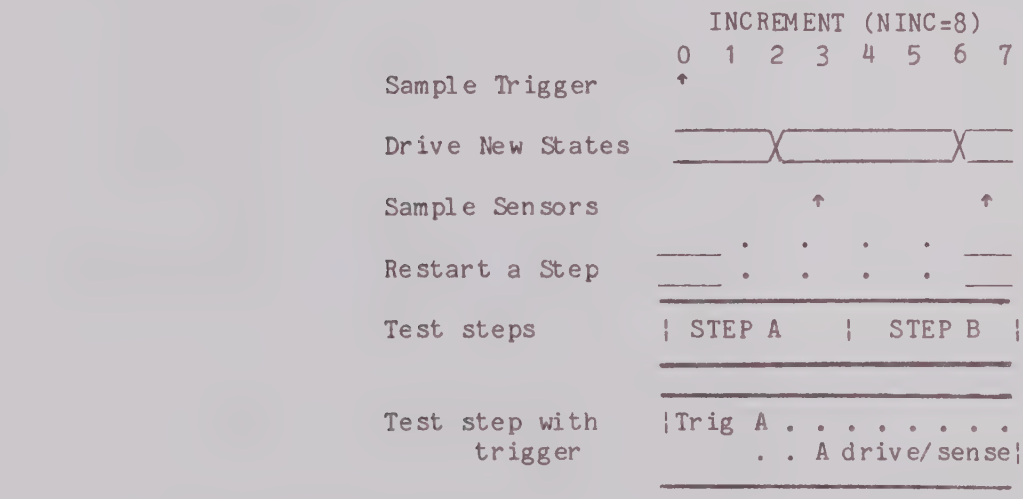
Using the first DST list, the first test statement always executes on increments 3 and 4; it cannot execute on 1 and 2 because no ED is present. Thus it is always suspended for at least 1 drive-sense cycle.

Using the second DST list, the first test statement executes immediately if the trigger condition is met, and the second statement executes immediately thereafter. If the trigger condition is not met, the first step is suspended for an odd number of drive-sense cycles, since it cannot execute on the second drive-sense cycle.

Using the third DST list, the first test statement can execute immediately if its trigger condition was met during the previous test step. The trigger sample coinciding with the next statement has no effect, since this second statement specifies no trigger condition.

DST Example: SET CLOCK . . . DST(T0,D2,S3,ED6,S7)

T0 means: Sample the trigger nails at increment 0
D2 means: Apply this test step's driver states at increment 2
S3 means: Sample for this step's sense states at increment 3
ED6 means: Re-enable driving and sensing; execute the suspended test step, applying the step's driver states at increment 6. A test step whose trigger condition has been met at the most recent T event executes immediately, without being suspended.
S7 means: Sample next step's sense states at increment 7



Note in this example that 2 test steps are executed for each pass through the timing list. Further, the trigger nails are sampled only on the first of these 2 steps, and if testing stops because a trigger condition is not met, this first test step then becomes aligned with the second half of the D/S list when testing resumes. (In effect, the test step pairs are shifted 180 degrees in phase relative to the drive-sense pairs.)

VCDL=volts Clock driver logic low voltage
VCDH=volts Clock driver logic high voltage
VCST=volts Clock synchronization high-to-low threshold voltage.
Values must be assigned for each of these terms, but they can also be specified in a prior ASSIGN LGC statement.

Range: -6 V to + 12 V. Resolution: approximately 30 mV

Default: values assigned by the most recent ASSIGN LGC statement

HISLEW/LOSLEW — Used to set the slew rate of the clock drivers (slew rates of test nail drivers are fixed). HISLEW gives about 250 V/us at a typical nail; LOSLEW gives about 125 V/us.
Default: LOSLEW

ON/OFF — Used to control the activation of the clock driver nails. ON commands the clock module to connect the clock drivers and start driving the UUT immediately. The clock drive signals will remain on and connected from this SET CLOCK statement until CLEAR CLOCK is executed or until the end of the test program.

OFF (default) causes the SET CLOCK statement to set up the clock module parameters but leave the clock driver nails disconnected until the next BURST or BUSTEST statement. The clock drivers then remain connected and driven during every succeeding burst, until CLEAR CLOCK or end of program.

Examples

1. SET CLOCK AT(5) F=1MEG NINC=4;

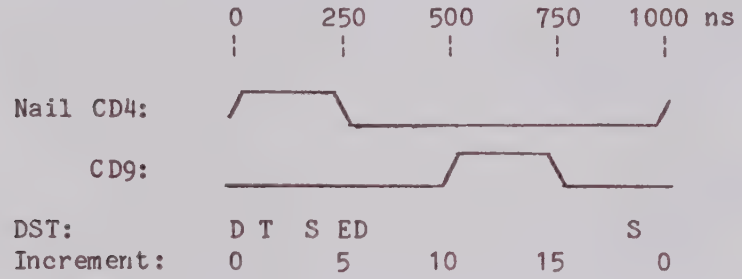
Meaning: At clock nail 5 apply a 1-MHz clock signal during each subsequent BURST or BUSTTEST section. Define 4 timing increments. Use all default timings. Defaults produce the waveform shown below with drive strobes (D) on each of the clock transitions, i.e., at 0 and at 500 ns. Sense strobes (S) occur immediately before the driver strobe, at about 450 and 950 (= -50) ns. Trigger sample (T) is at least 60 ns before the each drive strobe; in this case at about 440 and 940 (= D-60) ns. Enable drive (ED) is assigned to each drive strobe. The values of VCDL, VCDH, and VCST are taken from the most recent ASSIGN LGC statement or a previous SET CLOCK statement. This SET CLOCK can thus be represented as

```
SET CLOCK AT(5)
F=1MEG                  /*TINC = 1/(F * NINC) = 250 ns */
NINC=4                  /*PERIOD = 1/F = 1000 ns */
SIGNAL(CDA=L0,H2)
/* Default DST cannot be defined exactly; it is approximately
        ED1 = 0#ns    ED2 = 500#ns
        S = 450#ns, 950#ns
        T = 440#ns, 960#ns */
/* VCDL= from ASSIGN LGC */        0       500 ns      1000 ns
/* VCDH= from ASSIGN LGC */        ¦         ¦           ¦
/* VCST= from ASSIGN LGC */
OFF;                          CD5:  \_____/‾‾‾‾‾\
                          Increment:  0    1    2    3    0
                                      ED     TS ED     TS ED
```

2. SET CLOCK AT(4:9)
   TINC=50N NINC=20
   SIGNAL(H0,L5 : H10,L15)
   DST(D0,T2,S4,ED5,S19)

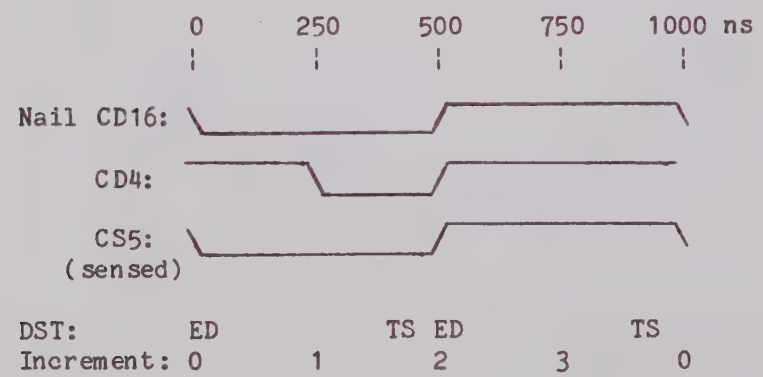
Meaning: Drive clock nails 4 and 9 with clock waveforms A and B, shown below. The waveforms are defined in the SIGNAL parameter by 2 transition lists. The first list (H0,L5) defines the waveform for clock A; the second list defines clock B. Since there are 20 increments (NINC) of 50 ns each (TINC), the period of the timing matrix is 20 * 50 = 1 us. The basic frequency is 1 MHz. The test rate, however, is 2 MHz because the driver-sensor timing list contains 2 drive-sense pairs.

Before a test containing trigger terms executes, it examines the trigger nail states stored from the most recent trigger strobe (T). Hence, a test step that falls on the first drive-sense pair in this DST list does not use the trigger nail states sampled at the time of the step, but rather the states that were sampled 2 steps earlier.



3. SET CLOCK AT(CDA=16 : CDC=4 : CSYNC=5)
   TINC=250N NINC=4
   SIGNAL(CDA=L0,H2 : CDC=L1,H2 : CSYNC=L0,H2)
   ON;

Meaning: Clock waveform A is supplied to clock nail 16 and clock B is supplied to clock nail 4. The ON parameter means that these clock signals are supplied as soon as the test program reaches the SET CLOCK statement, not just in BURST sections. There are 4 increments of 250 ns each for a timing period of 1000 ns. The DST defaults to the transitions of CDA to give ED1 at 0 ns, ED2 at 500 ns, S at 450 ns and 950 ns, and T at 400 ns and 900 ns.



Rules and Considerations

1. There should always be an equal number of drive (D) and sense (S) strobes in a timing list. If there are fewer sense strobes, the missing sense strobes are supplied at the default times relative to the following drive strobes. Fewer drive strobes than sense strobes are not permitted.

2. The trigger voltage threshold (VTT) is defined as a BURST or ASSIGN LGC parameter, not in the SET CLOCK statement.

----------SET LGC----------

General

SET LGC statements are used to control the terminating resistors on the driver/sensors and to specify state changes in the logic drivers, which cause corresponding state changes to the designated UUT input nails. The SET LGC statement activates the drivers to the specified states, but ignores the sensor outputs. Thus, the SET LGC command does not perform tests and, therefore, cannot pass or fail a test. An output nail list may be included, however, to alter the active nails for subsequent logic test statement.

Format

SET LGC (or $) nail-spec;

where: SET LGC Command keywords. Must appear first in the statement.

$ Symbol that may be used in place of the SET LGC keywords. Note that there must be a space between the $ and the nail-spec (e.g., $ IH(2), not $IH(2)).

nail-spec A list of input and output (optional) nail state specifications such as IH(1,4,5), OL(2,3), etc. (See Nail Specifications.) All input nails are driven to the indicated state, but no nails are tested. In FAST sections, the terms TH and TL are also recognized.

Examples

1. SET LGC IC(1-6) ID(7,8) IH(1-3) IL(4-6) OS(7,8) OH(7) OL(8);

   Meaning: Connect driver outputs to nails 1-6 and drive nails 1-3 high and nails 4-6 low. In anticipation of subsequent testing, disconnect the drivers from nails 7 and 8 and change the expected state of nail 7 to high and nail 8 to low.

2. $ IH(3);
   $ IL(3);

   Meaning: Generate a single pulse on nail 3.

3. $ PB(1);
   ID(1) OS(1) OM(1);

Meaning: Attach a pull-up and a pull-down resistor to nail 1 as shown in Figure 6-23. The driver/sensor system connects a 3.3-K resistor to the high driver reference voltage (e.g., +5 V) and a 2.2-K resistor to the low driver reference voltage (e.g., 0 V). Assume the tri-state driver on the UUT is being gated off (high impedance state) and is not terminated on the board. With the driver off, nail 1 should go to approximately +2 V (mid-state) due to the 3.3 K/2.2 K voltage divider. If, however, the driver output is shorted, nail 1 will go to 0 V, indicating a failure. Note that the PB(1) specification must appear in a separate SET LGC statement and cannot be combined with other nail-specs.

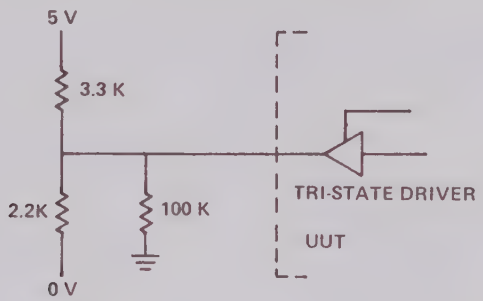


Figure 6-23. Sample Tri-State Bus Test.

Rules and Considerations

1. SET LGC changes the specified inputs and outputs but does not test any nails.

2. SET LGC can only be used in a burst.

3. There must be a space between the $ and the nail-spec.

4. Since a SET LGC statement always passes, it is not recommended that SET LGC be used with any conditional statement requiring a PASSES or FAILS status (e.g., GOTO, GOSUB).

5. Specifications for the terminating resistors PU, PD, PB and PN can only be used in a SET LGC statement that has no other type or nail-specs.

6. A SET LGC statement specifying a terminating resistor (PU, PD, etc) develops no pass/fail status of its own and leaves the current pass/fail status from the last logic test unchanged. Mid-state sensing (OM) is not available at SpeedPlus nails.

---------SUBROUTINE/END SUBROUTINE---------

General

Like non-burst subroutines, burst subroutines are initiated by a SUBROUTINE statement and terminated by an END SUBROUTINE statement. However, burst subroutines are called by GOSUB rather than CALL and they take no argument list. All burst subroutines must be defined at the end of the burst. The CLEARed state (i.e., disconnected, ignored and low) of all nails is assumed at the start of every subroutine.

SUBROUTINES cannot be called or declared within FAST sections.

Format

```
SUBROUTINE name;
    '
    '
    '
subroutine-statements
    '
    '
    '
END SUBROUTINE name;
```

where: SUBROUTINE — Keyword followed by the name of the subroutine indicates the start of the subroutine.

name — Name of the subroutine.

END SUBROUTINE — Keyword followed by the subroutine name; indicates the end of the subroutine.

Example

```
1.  ASSIGN LGC LVLA(1-7) VIHA= . . . . ;

    IC5: BURST IST= . . . . ;
         TEST LGC . . . . ;
         GOSUB PULSES FAILS;
         SET LGC . . . . . ;
             '
             '
         SUBROUTINE PULSES;
         IC(3,5,7) IH(3);
         IL(3) IH(5);
         IL(5) IH(7);
         IL(7);
         END SUBROUTINE PULSES;
             '
             '
         END BURST;
```

Meaning: If the TEST LGC statement fails, GOSUB will cause a branch to subroutine PULSES. At the start of the subroutine, nails 1-7 are all disconnected and low. Nails 3, 5, and 7 are then connected and pulsed as indicated, while nails 1, 2, 4 and 6 remain disconnected and low. When the subroutine is finished, program control is transferred to the SET LGC statement following GOSUB and the nail states resume their previous states prior to the subroutine.

Rules and Considerations

1. Subroutines must be defined at the end of a burst just before the END BURST statement.

2. At the start of each subroutine, all active nails (defined by the ASSIGN LGC statement ouside the burst) are disconnected, ignored and low.

3. No arguments can be passed to a subroutine called by GOSUB.

4. The END SUBROUTINE statement cannot be labeled.

----------TEST CLOCK----------

General

The TEST CLOCK statement is provided to allow the optional SpeedPlus clock module to achieve synchronization. The statement will test for synchronization and continue when synchronization (with the DUT node named in the CSYNC term in SET CLOCK) is achieved or when MAXTIME has been exceeded, whichever occurs first.

Format

```
TEST CLOCK
 MAXTIME=seconds
 FAIL(fail-bit-no.)
 [ error-processing ;];
```

where: MAXTIME — keyword for timeout. Range: 0 to 1.99 seconds with a resolution of 30.5 us. Default: 0 sec.

seconds — length of the timeout or maximum wait for synchronization to occur.

FAIL — keyword for failure indication

fail-bit-no. — number of bit in the FAIL bit-string that will be set if the timeout expires (synchronization is not achieved in the allowed time).

[error-processing] — any number of general-purpose test statements that execute if the timeout expires.

Example

1.
```
TEST CLOCK
    MAXTIME=0.5
    FAIL=13
    [WRITE 'SYNCHRONIZATION FAILED ON U13';]
    ;
```

Meaning: Wait up to one-half second for the SpeedPlus clock module to synchronize (presumably to U13). If synchronization is not achieved in this time, set FAIL string bit 13 and write the message to the console.

----------TEST LGC----------

General

TEST LGC is the basic digital test statement. Execution of this command drives all designated input nails to their specified states and tests all active inputs and outputs. As shown in Figure 6-24, a valid result occurs whenever any one of the following conditions exists:

1. The UUT nail voltage is more positive than the ASSIGNed VOH threshold and a high nail state (IH or OH) is expected.

2. The UUT nail voltage is more negative than the ASSIGNed VOL threshold and a low nail state (IL or OL) is expected.

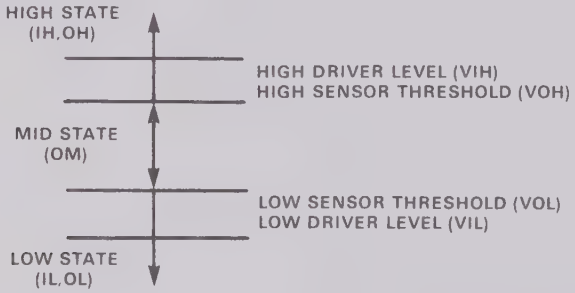


Figure 6-24. Driver and Sensor Reference Levels.

3. The UUT nail voltage falls between VOL and VOH and a mid-state (OM) is expected (not available on SpeedPlus nail).

If the sensed voltage level at any nail differs from the expected state of that nail, or if the command cannot be executed for any reason, a FAIL bit is set, if specified in the statement.

Format

TEST LGC (or null) nail-spec sequence-control-spec FAIL(n);

where:

TEST LGC (or null)	Command keywords. May be omitted since TEST LGC is the default statement within a burst. If it is used, the words TEST LGC must appear first in the statement.
nail-spec	A list of input and output nails such as IH(1-3), OL(4,5), etc, as described in nail Specifications. In FAST sections, TH and TL are also recognized.
sequence-control	Optional. Allows this statement or group of statements to be repeated under the specified conditions. (See Loops.)
FAIL(n)	Optional. Fail vector bit n. Set to 1 if a failure occurs. If n is a null (i.e., FAIL( )), then no fail bit is set. If FAIL(n) is not specified at all, then FAIL bit 1 is set when a failure occurs.

Example

1. IH(1-3) IL(4,5) OH(6,7) OL(8-10) FAIL(3);

   Meaning: Drive nails 1, 2 and 3 high and nails 4 and 5 low. Check all active nails for their designated state, including nails 1, 2, 3, 6 and 7 for a high state and nails 4, 5, 8, 9 and 10 for a low state. If an error is detected, set FAIL bit 3. (After the burst, the fail bit could be used to cause an error branch, such as IF FAIL(3) THEN BRANCH ERR3;.)

Rules and Considerations

1. TEST LGC can only be used in a burst or bus test section.

2. A TEST LGC statement will test all active nails, not only those specified in the TEST LGC statement, unless OI or ID has been used to disable sensing or driving at specific nails.

3. Each IH or IL is accompanied by an implicit sense command for the state of the driven nail, unless input sensing is explicitly disabled with II(<nail list>). A driven nail that fails to reach its programmed state by the time of the sense strobe causes a failure at that nail. IS(<nail list>) re-enables sensing of the specified driven nails.

Glossary of Terms – Appendix A

Term	Meaning
ARITHMETIC CONSTANT	A floating-point number up to 7 digits long. It may contain a minus sign, decimal fraction, and scaling factor or exponent. The exponent must be equal to or less than 30 (30th power). Examples: 256.0052, .1667, 47.P, 1.3U, 1.3E-6
BIT STRING	A sequence of bits, as used for example, in the fail vector.
CHARACTER STRING	A series of ASCII characters that may include alpha-numerics, punctuation marks, special symbols and white space characters (i.e., space, tab, carriage return). Examples: THIS IS A CHARACTER STRING RESISTOR R21 FAILED
DEVICE NAME	An alphanumeric identifier for a device. It is always terminated by a colon (:). Most monitor options require that the device name be specified. For the few options in which it is allowed, however, SY: (typically DK0: or DM0:) is the default disk device. TI: Video Display and Keyboard LP0: Strip Printer LP1: Line Printer DK0: to DK3:, or DM0: to DM3: Disk surface DY0: and DY1: Floppy disk surface To make device-naming less system dependent, logical device names may be substituted as shown below for physical device names. Refer to Test Set Preparation, Section 3, for more information.

Logical Device Name	Physical Device Name
LB0:, SY0:, MD0:,1:, LB1:,2:	DM0:, DK0:*
LB1:,2:*, AL0:, SR:	DM1:, DK1:*
SC0:, AL0:*	DM2:, DK2:*
SC1:, AL1:*	DM3:, DK3:*

*=2270 only

Note: The suffix 0 may be omitted from the device name (e.g., LB: for LB0:, SY: for SY0:, etc).

ESCAPE SEQUENCE	This term is used to refer to special characters bracketed by % signs and used in READ and WRITE statements to specify data formats or to indicate special "invisible" characters in a character string. %S%, %R%, %F%, %E%, and %G% are escape sequences that define the data formats.

Term	Meaning
ESCAPE SEQUENCE (cont)	%LF%, %CR%, %NL%, %011%, etc are escape sequences that define non-printing ASCII characters. Other types of escape sequences of the form ESC: A, ESC: B, etc may also be used in a WRITE statement to control the operation of the display and keypad indicators. (Refer to WRITE statement description.)
FILE NAME	File names are specified in terms of 4 distinct identifiers in the following format: device:name.ext;version where: device — Any one of the designators listed under LOGICAL DEVICE NAME or name — User-assigned file name. Up to 9 alphanumeric characters long. ext — File extension. Up to 3 alphanumeric characters long. version — An octal number automatically updated by the system each time the file is opened. Indicates the version of the file. The default value is the latest version: Example: SCO:BOARD1.TPG;12 Specifies version 12 (octal) of file BOARD1.TPG located on device SCO:. NOTE The file name for a test program that is to be used with the automatic fixture identification option must contain the decimal equivalent of the wired ID card. This feature suppresses leading zeros, so that a fixture wired for 0001 will cause the system to look for test program 1.OBC. (Refer to Section 4 of the System Operation manual.)
FILE-STRUCTURED DEVICE	A system device providing for random-access storage and retrieval of files such as the disk system and floppy disk devices. Devices that are not file-structured include video display, strip printer, and line printer.
FLOATING-POINT NUMBER	(See ARITHMETIC CONSTANT)
FORMAT STRING	A character string with or without embedded escape-sequences (%S%, %011%, etc), used in a READ or WRITE statement. Examples: '%011% TEST COMPLETE %NL%' 'RESISTOR R%F% FAILED'

Term	Meaning
IDENTIFIER	A string of alphanumeric characters that form the name of a variable or language element. An identifier must begin with a letter to distinguish it from a constant. Identifiers can be up to 10 characters long.
INTEGER	A string of digits with no fractional part. May contain scaling factor (see definition). The maximum integer allowed is 32767. Examples: 0, 43, 16384, 15K or 15000
LABEL	An optional identifier up to 8 characters long, used as a statement address. A label may begin with a letter or number and cannot be a system keyword. (Refer to para 2.5.)
OPERATORS	Operators are special characters that have some significant meaning in the test language. There are several types of operators: Arithmetic: Shown in order of precedence: Operator — Meaning * — multiply / — divide + — plus - — minus Relational: Two sides are compared. > more positive than (less negative than) >= more positive than (less negative than) or equal to < more negative than (less positive than) <= more negative than (less positive than) or equal to = equal to <> not equal to Character Strings: !! concatenate 2 character strings (i.e., link 2 character strings together).
RECORD	A line of data terminated by a line-feed character (octal 012).

Term	Meaning
SCALING FACTOR	Letter suffix representing a power of 10. Scaling factors are not allowed in Run-Time System operations such as Read. The allowable scaling factors are: MEG = $X10^{6}$ K = $X10^{3}$ M = $X10^{-3}$ U = $X10^{-6}$ N = $X10^{-9}$ P = $X10^{-12}$ E = Base 10 exponential form. The letter E is followed immediately by the exponent (preceded by a minus sign (-) if a negative exponent is being specified). Examples: 52E-4 = 0.0052 52E4 = 520000.
SOURCE FILE	Any file written and stored in test language form.
STATEMENTS	Test language statements may be either simple or compound. A simple statement is a single, stand-alone statement with command and parameter specifications. A compound statement is a sequence of zero or more statements delimited by brackets ([]). The syntax of a compound statement is identical to that of a simple statement, so that compound statements may be used anywhere simple statements are used. Typically, compound statements may be used in flow-control constructs (e.g., to define the statement sequence in a WHILE-DO loop) or as an [error- command] specification in an instrumentation statement.
VARIABLE	A DECLAREd name representing a data storage space. The variable name can be up to 10 characters long. The value can and typically does change during program execution. The type of each variable (i.e., floating point, bit string, character string, local, global, etc) is defined in the DECLARE statements at the beginning of the program (see DECLARE statement description in para 6.2). A program can contain any number of variables.
WHITE SPACE	Non-printing characters such as spaces, tabs, carriage returns, or comment strings.

Instruction Summary – Appendix B

ASSIGN LGC	Defines logic voltage levels for digital testing.
BITCLR	Resets a specified bit in a bit string.
BITSET	Sets a specified bit in a bit string.
BRANCH	Causes an unconditional transfer to a specified statement.
BREAK[1]	Forces an early exit from a burst.
BURST[1]	First statement in a digital test sequence (burst). Defines timing and other parameters used in the burst.
BUSTEST	Used to test buses and to isolate the source of bus failures.
CALL	Causes a program transfer to the indicated subroutine.
CLEAR	Resets the specified instrument to its initial condition.
CLOSE	Closes the specified I/O channel.
CONCATENATE	An operator (!) used to append one character string to another.
DATE( )	Function used to retrieve system date and time.
DECLARE	Defines all user-named variables and subroutines in the program.
DELAY	Suspends test program execution for the specified time interval.
END	Last statement in a test program.
EXIT	Stops further testing and clears the test hardware.
END BURST[1]	Last statement in a burst.
END SUBROUTINE	Last statement in a subroutine.
EXITIF[1]	Used in a burst to specify additional loop exit conditions.
FLOOP/END FLOOP[1]	Used to loop within a FAST section.
FNAILOFFSET	Used to relocate fast nail lists.

[1] Can be used only in a digital burst.

GOSUB[1]	Used in a burst to call up a subroutine.
GOTO[1]	Used in a burst to transfer program control to a specified label.
IF-THEN-ELSE	Specifies conditional program statement execution.
INDICT	Specifies an IC to be probed.
KEYBD( )	Function used to sense keyboard (or keypad) strokes.
LENGTH	A function that determines the actual number of characters in a character string.
LET	Assigns a value to a variable or defines that variable in terms of another variable or expression.
LOAD/UNLOAD	Defines variables or subroutines that will exist for the test program from the time the program is loaded until it is unloaded.
LOOP/END LOOP[1]	Used for looping within a burst.
LOOPIF[1]	Used in a burst to specify additional looping conditions.
MEAS	Causes the instrument to measure a response (typically from a UUT), report a failure if detected, and store the response for future access by the program.
NAILOFFSET	Used to relocate pin lists.
ONABORT	Defines operations that the system will automatically perform upon detecting an abort condition.
OPEN	Opens an I/O channel for subsequent data transfers via READ or WRITE commands.
OR[1] or	Used in a burst to specify alternate conditions for repeating terminating a loop.
PAUSE	Temporarily suspends program execution until the START/CONTINUE button is pressed.
READ	I/O command that directs data from the specified OPENed channel to the indicated variable(s).
RETURN	Can be used to cause an immediate exit from a subroutine.
SET	Forces the specified instrument into a particular configuration.

[1] Can be used only in a digital burst.

SUBROUTINE	Defines the first and last statements in a subroutine.
SUBSTR	A function used to define a character substring.
TEST	Causes the instrument to check a response (typically from the UUT), compare it against predefined limits, and indicate a pass or fail status depending on the results.
TIME( )	Function used to retrieve system time.
USE	Defines diagnostic options for subsequent instrumentation statements.
WHILE-DO	Allows a program loop to be executed as long as the specified condition is true.
WRITE	I/O command that directs data to the specified OPENed channel.

Instruments

ACZ - AC impedance measurement unit
ARITH - Arithmetic test
DCM - DC measurement unit
DCS - DC source
I488 - IEEE (or IEC) bus
LGC - Digital driver/sensors
MUX - Instrument multiplexer
OPENS - Open-circuit detector
PIO - Parallel I/O (Relay Driver board)
RM - Ohmmeter
SCAN - Analog pin scanner
SHORTS - Shorts detector
STM - Self-test module
TPS - Triple power supply
UPS - Universal power supply

Preparing External Routines – Appendix C

C.1 INTRODUCTION

The following paragraphs describe how an external subroutine interfaces with a 227X test program and how a programmer must code a routine that is to be called from a 227X test program. This information is directed at programmers familiar with RSX-11M and MACRO-11 and assumes some understanding of RSX-11M system calls, although the programmer will not need to invoke any of these directly.

The 227X test language allows calls to both internal and external subroutines. An external subroutine, however, is not simply a procedure that is externally compiled and linked to the main test program. Instead, it is an independent program, written in MACRO-11 or a high-level language other than 227X test language, which is assembled, task-built, and installed on the 227X as an independent machine-code program or task. External routines thus allow a system programmer to define as a 227X external subroutine any action that is possible under the 227X's RSX-11M operating system and, if necessary, to extend the capabilities of the 227X language as desired, by defining new operations as external subroutines. The preparation of an external subroutine requires a facility equipped with the DEC RSX-11M Taskbuilder, a MACRO-11 assembler, and a compiler if a high-level language is used.

CAUTION

The 227X EXTERNAL SUBROUTINE interface is supplied as an aid for experienced PDP-11 programmers. External subroutines should be carefully verified before use, and all system software should be backed up before such programs are tested. GenRad will be able to support only the interface routines supplied as part of the 227X software package.

C.2 THE EXTERNAL SUBROUTINE AS AN INDEPENDENT TASK

C.2.1 Differences between Conventional and 227X External Routines

When a 227X test program calls one of its own subroutines, it effects a jump to the address of the routine, passing parameters via the stack. Hence, an internal 227X subroutine call is very much like a function or procedure call in most other high-level languages. Many high-level languages also allow calls to routines that have been separately compiled and linked to the main program's code by the compiler or a linkage editor. These routines must usually be declared within the main program and marked with a keyword such as "EXTERNAL," which later tells a link editor to substitute the address of a routine found in another file or module.

Even though the actual code for an 'external' routine of this kind is not originally present in the source code of the program, it is possible to call the routine by virtue of its being incorporated into the machine code at a later time by the linkage editor. Therefore, an 'external' call of this sort actually turns into a call that is still purely internal to the machine-code program.

C.2.2 227X External Subroutine Calling Mechanism

A mechanism for making calls to external subroutines has been incorporated in the 227X test language which, to the 227X programmer, appears to function like this conventional form of 'external' call. Once a subroutine has been made available on the system, the test programmer needs only to know its name and its calling syntax, as with a 227X local subroutine.

The actual implementation of the 227X external subroutine call is, however, quite different from the conventional calling mechanism described above. No external code is included in the compiled 227X test program. Instead, when an external routine is to be called, a 227X CALL statement causes the 227X system to issue a series of instructions to the operating system which start the external routine and mediate communication with it.

While, for the test language programmer, an external routine appears to be called in exactly the same way as an internal routine, the calling mechanism is actually far more complex. It may be compared to the mechanism that allows the test program to communicate with peripherals (which, like the external subroutine, cannot be addressed directly). During execution, the compiled 227X test program carries out all of its actions (i.e., is interpreted) by calling a large, lower-level program called the 227X run-time system. The run-time system, in turn, must call on the computer's operating system for such general services as file management and I/O operations. Even a simple action, such as writing a message to the terminal, may require a call to the operating system. A statement such as

```
WRITE 'SHORTS FAILED'
```

executes first as a call from the compiled test program to the 227X run-time system, which in turn issues a special kind of call statement called a directive and passes the message to a terminal driver routine in the operating system.

A CALL to an external subroutine executes in a similar way, as shown in Figure C-1. The CALL statement compiles as a jump to a rather large routine in the 227X run-time system, which then sets up an interface to the external subroutine via the operating system. This run-time routine does a number of things: It first asks the operating system to load a program (with the specified subroutine name) installed on the system; it then sends and receives the data specified in the CALL statement via operating system directives that communicate with the external routine in any number of individual transactions. Finally, it returns control to the next test program statement. (It also detaches the user's terminal, issuing QIO IO.DET, to allow the external subroutine to address the terminal directly, if necessary, as well as through the 227X message file.)

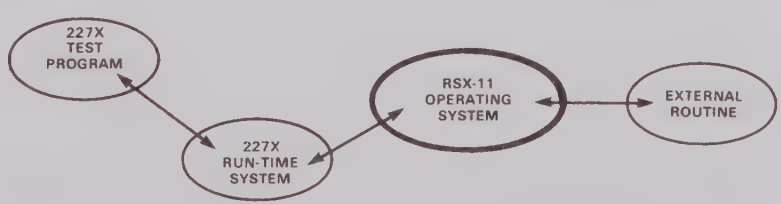


Figure C-1. 227X Interface Relationships.

The called routine must mirror the interface protocol of the 227X run-time system. Accordingly, writing an external routine that interfaces correctly with the 227X CALL statement requires more programming effort than writing a conventional subroutine. Specifically, the programmer must:

a. Write a program in MACRO-11 or a high-level language other than 227X test language to perform the desired operation.

b. Embed or call a number of (supplied) routines written in MACRO-11 which allow the program to communicate with the operating system and, through it, with the 227X run-time system.

c. Compile and/or assemble, taskbuild, and install the program so that it may be executed by a CALL statement in a 227X test program.

C.3 THE EXTERNAL SUBROUTINE INTERFACE PROTOCOL

Since the external subroutine is in reality an independent machine-code program (task image) that normally resides on disk and has no address known to the test program, the 227X test program cannot communicate with it directly. Instead, both the test program (via the run-time system) and the external subroutine must send and receive parameters and control messages through the operating system, and each must know what it must send and what it can expect to receive at each step.

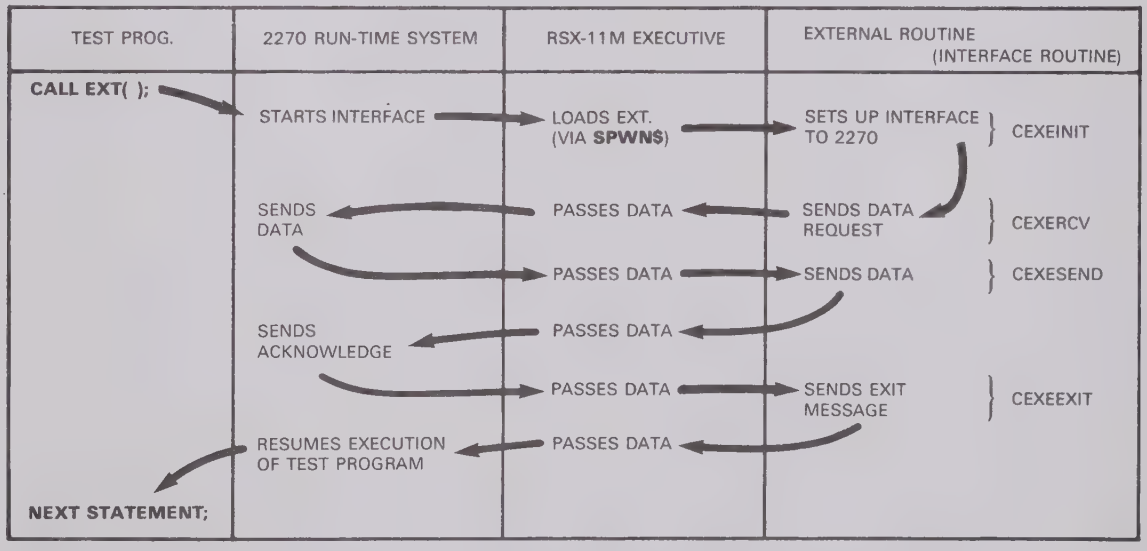


Figure C-2. External Subroutine Interface Protocol.

The 227X external CALL statement initiates a series of events which require a series of appropriate responses by the called routine. These responses must be coded in the external routine in the form of MACRO-11 calls to RSX-11M system directives, supported by some housekeeping operations required to pass data to system event flags and to and from the directives. All the necessary interface operations have been coded in four MACRO-11 routines (Cexeinit, Cexercv, Cexesend, and Cexeexit) supplied in the file DK2:EXECUTE.MAC. The author of an external subroutine can accomplish all the necessary communication with the 227X run-time system (and thus with a test progam's CALL statement) by simply calling these routines from appropriate places in the external subroutine. Figure C-2 shows in a simplified way how the routines function in passing data and processor control back and forth between the 227X run-time system and the called routine.

When a 227X test program CALLs an external subroutine, the 227X run-time system first issues a system directive (SPWN$) that loads the routine -- as a separate task, with its own memory space -- into core. The executive then starts the external subroutine, whose first responsibility is to establish communication with the calling 227X task by fetching the name of the 227X run-time task and clearing an event flag. This can be done by calling Cexeinit. The external subroutine must then call one of the other interface routines, which will send a data packet back to the 227X task. If the data packet sent by the routine contains the message "finished" (normally sent by Cexeexit), the communication ends here and control returns to the 227X text program. (Sending the "finished" code EXE$DONE (value = -1) via Cexesend or Cexercv will require directive modifications and is not recommended.)

Otherwise, the external subroutine must next send either

1. A request for one or more parameter values from the test program's CALL statement (Cexercv)

or

2. One or more values to be assigned to parameters in the test program's CALL statement or to be sent to files MESFILE or LOGFILE (Cexesend).

The external subroutine can request or send any of three data types defined in the 227X test language: FLOAT, BSTRING, and CSTRING. After it has received data from the external subroutine or a request for data, the 227X run-time task returns a reply, either the requested data, a "data received" message, or an error message. Any number of transfers to and from the CALL parameters can be made; the 227X test program, however, cannot perform any processing while the external subroutine is executing. Once the external subroutine issues a message that it has finished (Cexeexit), the 227X run-time system again assumes control and the executive removes the external subroutine from its list of loaded tasks. Hence, the external subroutine is effectively no longer in memory, and each new call requires that it be loaded again from the disk where it is stored. External subroutine calls therefore entail a significant time overhead (typically 40 ms) that is unrelated to the execution time of the routine itself. This overhead can be reduced somewhat by installing the external routine permanently in memory with the MCR command FIX. Refer to the DEC MCR Operations Manual for a description of the FIX command.

C.4 CREATING AN EXTERNAL ROUTINE

Each step in the interchange described in para C.3 requires a call to one of the four MACRO-11 routines (with their associated subroutines) named in Figure C-1. These can be made available to the program by linking it to the file DK2:EXECUTE.MAC. These routines can be called without difficulty from a MACRO-11 program, but high-level languages will usually impose calling conventions different from those expected by the routines. It is the programmer's responsibility to make all necessary accommodations. The external subroutine program must then be assembled or compiled, taskbuilt, and installed on the system by a privileged user.

C.4.1 Programming Guidelines

To use the routines in DK2:EXECUTE.MAC, the programmer must satisfy certain requirements imposed by the 227X's CALL statement protocol and by the interface routines. These requirements are:

1. All communication with the 227X test program is effected by calling the routines in DK2:EXECUTE.MAC (or equivalent routines).

2. Control must be passed to all routines in DK2:EXECUTE.MAC with a JSR instruction or an equivalent stack-oriented procedure or function call. (Return is via RTS PC.) Return status (NO ERROR = 1, ERROR = 0) is put in register 0, which should be checked for error status after the call. If an error occurs, the error code is put in the global word "Cexestatus." Error codes are RSX-11M directive error codes (-1 through -99) or special EXECUTE error codes (-100 through -103) defined in EXECUTE.MAC. Directive error codes are passed back to Cexestatus from the directive status word ($DSW) by the local subroutine err (see the DK2:EXECUTE.MAC source code). For an explanation of RSX-11M error codes, refer to the DEC RSX-11M Executive Reference Manual, Appendix B. The four external error codes are generated by the interface routines in EXECUTE.MAC and are defined in the examples at the end of this appendix. High-level language function or procedure calls may be used if the language compiler supports the declaration of external symbols. The routines in EXECUTE.MAC may need to be modified to obtain the proper interface.

3. Data buffers of adequate length must be set aside in the external routine. FLOAT variables will require two words (32 bits). BSTRINGs may be as long as 517 words but require at least three words. CSTRINGs may be as long as 517 words (1034 characters) but require at least one word (two bytes).

   FLOATs are in the usual PDP-11 floating-point format. BSTRINGs consist of one word giving the number of bits (in the string that actually follows), a second word giving the number of implicit leading zeros (to be inserted), and at least one word of data. CSTRINGs consist of at least one byte of data, terminated by one or two bytes of zeros (null byte) landing on a word boundary (even address). The addresses of the buffers are passed to Cexesend and Cexercv in a send or request block.

4. Send and request blocks must be declared in the external subroutine for informing the Cexercv and Cexesend routines about the locations and data types of the data to be passed to or from the 227X task. (They do not, themselves, contain the parameters being passed.) Send and request blocks must contain, in the order shown, three items of data about every variable to be sent or requested:

   1 byte containing the parameter's position in the CALL statement (1st, 3rd, etc, up to 255th) as an unsigned 8-bit value

   1 byte containing the 8-bit numerical code for the data type (FLOAT = 1, CSTRING = 2, BSTRING = 3) or for a special message destination (message file = -2, log file = -3, CSTRING is assumed as the data type)

   1 word containing the address of a buffer to store the data to be sent or requested.

   Any number of variables may be specified in a given send or request, provided that no more than 517 words are specified altogether (that is, no more than 517 words can be passed with a single call to Cexesend or Cexercv).

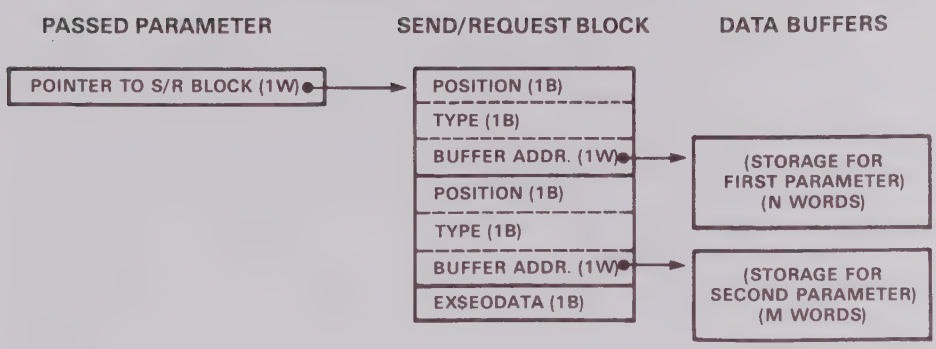


Figure C-3. Structure of Send/Request Block.

The block must end with the value EXE$EODATA (null). A pointer to the send or request block must be pushed onto the stack before Cexercv or Cexesend is called; the routines do not remove the pointer from the stack before control is returned to the calling segment. Figure C-3 shows the structure of a send/request block and its relationship to this pointer, as well as to the associated data buffers.

When Cexesend or Cexercv is called, only the variables identified in the send or request block whose address is on the stack will be sent. Thus, one may request the first and fourth parameters from the CALL statement at one time, and the second and fifth another time, by passing the addresses of two different request blocks.

5. A call to Cexeinit with an integer value on the stack must precede all other calls. Cexeinit must fetch the Rad50 name of the calling 227X run-time task (which is not necessarily the same under all circumstances), and it will normally do this by issuing a GMCR$ directive if it is called with a zero-value parameter. However, a high-level language or a larger MACRO-11 program may itself include the system command GMCR$ in its initialization routine, which means that the directive's buffer will be cleared and the Rad50 task name will no longer be available by the time Cexeinit is called. To allow for this possibility, Cexeinit can be called with a non-zero-value parameter; instead of obtaining the task name via GMCR$ as it normally does, Cexeinit will simply look for the Rad50 task name at the address supplied as the parameter.

   A call to Cexeexit must terminate the external subroutine and it must be called with a status word on the stack (indicating either successful completion or an error condition, e.g., EXE$TMD).

6. Parameters from the 227X test program's CALL statement are fetched with Cexercv. The address of a request block that describes all the items of data to be fetched (position in CALL statement, data type, address of local data buffer) must be pushed onto the stack before Cexercv is called. This address remains on the top of the stack after Cexercv has executed.

7. Parameters are sent to the 227X test program's CALL statement with Cexesend. The address of a send block that describes the data to be sent must be pushed onto the stack before Cexesend is called. This address remains on the top of the stack after Cexesend has executed.

The routines contained in EXECUTE.MAC can be called without modification from a MACRO-11 program or (as function calls) from a program written in C (Unix, Yourdon, or Whitesmiths compiler). Other high-level languages may require either changes to EXECUTE.MAC or special calling arrangements to retrieve R0 or to pass pointers to the send/request blocks properly. Consult your compiler documentation to find out how procedure and function calls are implemented and how symbolic references to external modules can be made. If possible, examine the object file listing (extension .LST) from the assembler to see how your calls compile.

Most high-level languages will require special attention to two requirements imposed by the EXECUTE.MAC routines. First, a data structure of the required form must be provided. In some high-level languages a send/request block can be constructed as a record (Pascal) or structure (PL/1, C), but in many languages an appropriate data structure may prove difficult to construct because it must contain the address of, or a pointer to, the data buffer used for each variable rather than the data buffer itself. (This ensures that a block always has the same structure, regardless of the format of the data to be passed.) In Fortran or Basic, for example, the block can be declared as an array of single precision integers. For each parameter to be passed, each odd-numbered array element will be assigned an integer value corresponding to the first two byte-length values (for the position and data type descriptors). The even-numbered elements must contain addresses of the corresponding data buffers. If the data elements are declared as named COMMON blocks or variables, a short MACRO-11 routine can be called to insert their addresses in the appropriate array positions. Alternatively, one can declare the entire send/receive block as a named COMMON block and alter the MACRO-11 code in Cexesend and Cexercv to load the addresses of the data buffers into the block with instructions of the form

```
MOV #BUFFR,BLK.PNTR
```

after the block has been passed from the high-level program.

Second, parameters must be passed properly to the EXECUTE routines. Languages like Basic and Fortran often place the address of a parameter block not on the stack but in a register, whereas the EXECUTE routines look for this address on the stack. Likewise, a stack-oriented language like Pascal may place a returned function value on the stack rather than in R0. One approach to the resulting interface problem is to write "shell" procedures which make the required accommodations before the EXECUTE routines are called. Again, however, it will probably be simpler to change the code in EXECUTE.MAC to correspond to your language's calling conventions.

One short example of an external routine is provided in Figure C-4.

C.4.2 Taskbuilding and Installing the External Routine

Under RSX-11M, assembled object modules are not directly executable. Instead, the taskbuilder utility, TKB, must link them with system directive libraries and a run-time system to produce an executable machine-code file, called a task image. The taskbuilder allows the programmer a wide range of options in specifying how an object file will be linked to other object modules as well as to the operating system and how it will be loaded. It also resolves global references among object modules so that a number of object files can be combined into a single task image, and it provides options for the final structure of the task image, which usually default to commonly required values. After assembly of both the file EXECUTE.MAC and the external routine's source

code, the executable task image will typically be created with a taskbuilder command line of the following form (for a program named Trigvals):

```
>TKB TRIGVALS=TRIGVALS,EXECUTE,LB:[1,1]LANGUAGE.LIB/LB
```

where the second TRIGVALS, EXECUTE, and LB:[1,1]LANGUAGE.LIB represent, respectively, the external routine object file (.OBJ), the assembled EXECUTE file (.OBJ), and the high-level language's object library (not required for MACRO-11). (The system object library is included automatically.) This taskbuilder command will usually suffice; however, you may need to override some of the taskbuilder defaults in cases such as these:

1. Heavy recursion: Use the STACK= option to extend the space allocated by the taskbuilder for the program stack.

2. Dynamic memory expansion: Use either the /CP command line switch to allow checkpointing for dynamic memory expansion by the operating system, or determine the name of the dynamic memory area and use the EXTSCT option to increase its size (e.g., EXTSCT=$$HEAP:).

3. Floating-point operations: Floating-point macros must be provided where floating-point operations are required; do not use /FP (since the 227X does not contain a floating-point processor). A package of MACRO-11 floating-point operations is supplied in DK2:EXEFF.MAC; they may be called as global subroutines with pointers to the floating-point parameters passed via the stack. Like DK2:EXECUTE.MAC, this file must first be assembled and then included in the taskbuilder command line to be incorporated in an external subroutine.

4. More than 6 logical units (external files or device interfaces): Use the UNITS= option to declare more Logical Unit Numbers (LUNs). Use the ASG option to assign LUNs to a particular disk drive or device.

5. More than 4 character files: It may be necessary to extend the size of the FCS buffer area with EXTSCT=$$FSR1:.

Consult the DEC RSX-11M Taskbuilder Manual, the RSX-11M Guide to Program Development, and the documentation for your programming language for detailed information on how to prepare and taskbuild a program for your system.

After object files, including EXECUTE.OBJ, have been linked to an external subroutine program (e.g., TRIGVALS.OBJ) by the taskbuilder, the resulting task is found in the task file (e.g., TRIGVALS.TSK). This machine-code program cannot be accessed by the 227X run-time system, however, until it is entered with an identifier of six characters or less in the executive's list of installed system tasks. The 227X system cannot call the task file directly. The external subroutine must be installed on the system by a privileged user, using an MCR command such as

```
>INS DK1:[1,123]TRIGVALS.TSK;4
```

which truncates the name to six characters and installs it as TRIGVA on the system. It must therefore be called under the name TRIGVA from a 227X test program, with the statement

```
CALL TRIGVA( <parameters> );
```

Like TKB, the install utility provides a number of options which may not always default to the programmer's liking. For example, it is not necessary to install the task under the name of the .TSK file. Consult the DEC MCR Operations Reference Manual for detailed information on the use of INS.

If the external subroutine is used regularly on a system, the boot-up command file should be edited to include the INStall command, so that the task will be installed each time the system is booted. Include the INStall commands in the file LB:[1,2]STARTUP.CMD.

To improve the calling time of a frequently used external subroutine, it may be loaded into a permanent location in core memory with the FIX command; this saves the time required to load the subroutine from disk, at the expense of dedicating a segment of memory. Add the FIX command to the boot-up file after the INStall command, in the form

```
FIX TRIGVA
```

A memory partition dedicated to a task with FIX can be freed with the MCR command UNFix; the task can be removed from the list of installed tasks with the REMove command.

C.5 PSEUDO-CODE SUMMARIES OF THE INTERFACE ROUTINES

The following representations of the interface routines assume a calling mechanism that pushes parameters onto the stack and returns function values or pointers in R0.

C.5.1 Initialization

The Cexeinit subroutine is called with a null address on the stack (one word). The external subroutine learns the name of the 227X run-time task, with which it must communicate, by fetching it with the directive $GMCR. If a non-zero address is passed, Cexeinit looks for the Rad50 name of the run-time system at the address on the stack.

```
ROUTINE Cexeinit (addr_of_taskname)
  BEGIN
    IF (addr_of_taskname = null) THEN
        fetch Rad50 RTS taskname; /* gmcr directive */
    ELSE
        copy Rad50 RTS taskname;
    specify receive Asynchronous System Trap (AST) routine;
    disable ASTs;
    clear event flag; /* use event flag #1 */
  END
```

C.5.2 Exiting

The Cexeexit subroutine must be called with a status word pushed on to the stack when the external subroutine has completed its function.

```
ROUTINE Cexeexit ( status )
  BEGIN
        put status in end-of-task block;
        send end-of-task block;
        terminate execute routine;
  END
```

C.5.3 Requesting Data

The Cexerecv subroutine is called with the address of a request block on the stack when data are needed from the 227X run-time system (RTS). The format of the request block is illustrated in the examples at the end of the appendix; it contains parameter descriptions and pointers to the data buffers.

```
ROUTINE Cexerecv (addr_of_request_block)
  BEGIN
        pack request block in send buffer; /* calls 'packsb' */
        send request;
        enable ASTs;
        wait for 227X RTS reply; /* use event flag #1 */
        clear event flag; /* set by receive AST routine */
        receive all data;
        disable ASTs;
        IF (length = 0) THEN
              fetch error code;
        unpack data;
  END
```

C.5.4 Sending Data

The Cexesend subroutine is called with the address of a send block on the stack when data are being sent to the RTS. The format of the send block is like that of the request block and is illustrated in the examples in para C.5.5.

```
ROUTINE Cexesend (addr_of_send_block)
  BEGIN
        pack send block in send buffer; /* calls 'packsb' */
        send data;
        enable ASTs;
        wait for 227X RTS reply; /* use event flag #1 */
        clear event flag; /* set by receive AST routine */
        receive acknowledge packet;
        disable ASTs;
        IF (error) THEN
              fetch error code;
  END
```

C.5.5 Send/Request Buffer Formats

The subroutine packsb is a local routine (it cannot be called by the user). When called by Cexesend or Cexercv, it uses the information in a send/request block to build a send/request buffer, as required by the system

communication directives, which is sent to the 227X run-time system. A pseudo-code representation is given here simply to describe the format of this send/request buffer.

```
ROUTINE packsb (addr_of_send/request_block, send_flag)
  BEGIN
        point to buffer+1;
        WHILE (not end of send/request block)
           BEGIN
                put argument number in buffer;
                put argument type in buffer;
                IF (sending data) THEN
                    BEGIN
                        make argument number negative;
                        put data in buffer;
                    END
           END
        store length of block in 1st word;
  END
```

C.6 A SAMPLE EXTERNAL ROUTINE

An example of an external subroutine is shown in Figure C-4. This program, RSHIFT.MAC, is written in MACRO-11 assembler. It performs a logical right shift on a 16-bit BSTRING, thus providing a function not available in 227X test language. Two parameters are passed: the BSTRING and an ASCII character from '1' to '9', giving the number of places to be shifted. If an out-of-range character is passed, an error message is sent directly to the user's terminal. The shifted bit string returns to the same parameter in the 227X CALL statement. A MACRO-11 programmer may take RSHIFT as a template for more complex routines.

RSHIFT must be declared in a 227X test program with the statement

```
DECLARE EXTERNAL RSHIFT(BSTRING A(16), CSTRING B(1));
```

and must be called with a statement of the form

```
CALL RSHIFT(A=STR1, B=STR2);.
```

Users of high-level languages must find out how their compilers allow links to external object modules, and then they must provide appropriate calls to the EXECUTE routines, as described in para C.4.1. After studying RSHIFT.MAC, a high-level language programmer should work out carefully how these calls can be implemented, paying particular attention to two crucial points. First, the status codes from the 227X run-time system and the EXECUTE routines are returned to R0 and cannot be recovered with some kinds of function call. A compiler that allows embedded assembly code may allow you to move the status value to a program variable; if necessary, an assignment statement can then move it to a function identifier or parameter. But, clearly, one could also move R0 directly to a position on the stack or to a parameter block where it would automatically be returned as a function value; the EXECUTE.MAC routines, modified to do this, could then be called as external functions.

The second crucial point concerns the construction of the send and receive blocks. Again, it is necessary to know how your compiler works and how it can be forced to produce the required data structure. It may be difficult to specify character (byte-length) constants of the required control character

values for the position and data-type codes in the send or receive block (in any case, they could not be printed), so another expedient is to declare variables and later to assign the required values to them. In the case of pointer constants, Pascal, for example, requires that they be created by means of the standard procedure "new." The object code must be examined to ensure that the data block produced by the compiler has the structure expected by EXECUTE.MAC.

All the functions in EXECUTE.MAC return a status value in the parameter "Cexestatus." In RSHIFT.MAC, the status value is simply returned to the 227X. This value can then be used as an error diagnosis aid, but it can also be used within the external subroutine program to direct it to repeat an attempt to pass data that has failed for system-dependent reasons, such as lack of available dynamic memory (standard executive error codes -1 or -4).

The program shown in Figure C-4 must be linked, after assembly, to EXECUTE.OBJ and installed on the system before it can be called as a 227X external subroutine.

```
;
;function:      show simple Macro-11 example of external subroutine
;

main::                                  ;<--entry
        clr     -(sp)                   ;pass null pointer
        jsr     pc,Cexeinit             ;initialize execute routine
        tst     r0                      ;error?
        beq     err2                    ;if yes, abort

        mov     #reqblk,-(sp)           ;pass pointer to request block
        jsr     pc,Cexerecv             ;request and receive data
        tst     (sp)+                   ;pop stack on return from Cexercv
        tst     r0                      ;error?
        beq     err2                    ;if yes, abort

        sub     #61,cstrg               ;convert char to numerical value
        cmp     cstrg,#11               ;unsigned range check
        bhi     err1                    ;if character out of range '1'
                                        ; to '9' abort, report type error

        clr     r1                      ;for use as loop counter
        movb    cstrg,r1                ;get count value into r1 and

1$:     ror     bstrg+4                 ;rotate the Bstring
                                        ; (data start in 4th byte of buffer)
        bic     #100000,bstrg+4         ;clear msb to simulate logical shift
        sob     r1,1$                   ;loop until zero


        mov     #sndblk,-(sp)           ;pass pointer to send block
        jsr     pc,Cexesend             ;send data and message
        tst     (sp)+                   ;pop stack on return from Cexesend
        tst     r0                      ;error?
        beq     err2                    ;if yes, abort,

        mov     #TRUE,-(sp)             ;else success!
        jsr     pc,Cexeexit             ;exit-->

err1:   mov    #EXE$BTY,Cexestatus      ;type of error in status word
err2:   mov     Cexestatus,-(sp)        ;abort and pass error status to 227X
        jsr     pc,Cexeexit             ;exit-->

        .end    main
```

Figure C-4. An External Subroutine in MACRO-11. (Page 1 of 2)

```
        .psect Cexestatus,ovr,gbl

Cexestatus:
        .blkb   2                       ;global execute status

        .psect  m.data
        .even
;
;       local data buffers
;
bstrg:  .blkw   3                       ;local storage for BSTRING
                                        ;(2 descriptor words, 1 data word)
cstrg:  .blkb   2                       ;local storage for CSTRING
                                        ;(1 character, 1 null)
        .even

message:                                ;ASCII message for mes file
        .asciz  /external task has completed/<12>
        .even

reqblk::                                ;request block
        .byte   1,EXE$BSTRING           ;requesting 1st argument, a Bstring
        .word   bstrg                   ;to be put in 'bstrg'
        .byte   2,EXE$CSTRING           ;requesting 2nd argument, a Cstring
        .word   cstrg                   ;to be put in 'cstrg'
        .byte   EXE$EODATA              ;end of data
        .even

sndblk::                                ;send block
        .byte   1,EXE$BSTRING           ;sending Bstring to 1st argument
        .word   bstrg                   ;value from 'bstrg'
        .byte   EXE$NOARG,EXE$WMES      ;writing string to message file
        .word   message                 ;address of message
        .byte   EXE$EODATA              ;end of data
        .even
```

Figure C-4. An External Subroutine in MACRO-11. (Page 2 of 2)

Set and Meas Commands on the I488 Bus – Appendix D

D.1 INTRODUCTION

To provide compatibility with earlier sofware releases as well as flexibility required for unusual applications, the 227X test language recognizes 2 special uses of the SET and MEAS commands for the IEEE-488 bus (referred to here as I488). For most purposes it will be found that READ ID=I488 and WRITE ID=I488 perform the required input and operations to I488 bus instruments. The SET commands, however, allow more exact control of I488 transactions and may therefore be useful for communicating with an instrument that does not implement a standard IEEE-488 protocol.

The I488 bus allows the bus controller (normally the 227X) to issue both bus management command and instrument messages. When a command is issued, the dedicated command bus line ATN is active, and the byte on the data bus is interpreted as part of a bus command. When the ATN line is inactive, bus data are regarded as messages between bus nodes or instruments. The SET I488 command allows strings of bus commands or instrument messages to be sent to the bus. SET I488 CMD OUT='command-string' sends the specified bytes (given as octal values in command-string) to the bus while holding ATN active, so that each character is interpreted as a bus command. Omitting the keyword CMD causes the SET command to hold ATN inactive, so that SET I488 OUT='message-string' sends the string of characters specified by message-string as an instrument message.

MEAS functions similarly to READ; however, only a single string variable is permitted as a parameter. Unlike READ and WRITE, MEAS and SET do not restrict the number of characters that can be sent or received with a single statement.

Format

```
SET I488 CMD OUT='command-string' [BRANCH statement;];
SET I488 OUT='message-string' [BRANCH statement;];
MEAS I488 INTO variable;
```

where:

CMD OUT=	Keyword indicating that the following command-string will cause the I488 Controller to activate the I488 ATN line. Bus data will then be interpreted as standard I488 commands. With CMD OUT=, SET does not recognize the I488 command mnemonics (refer to para 6.3, I488, for use of mnemonics directly in SET I488).
OUT=	Keyword indicating that the following message-string will be sent to the addressed device as a character-string over the I488 data bus. ATN is not active.
INTO variable	Specifies a string variable into which the received (MEASured) data will be stored.
[BRANCH-statement;]	The program will branch as specified if the time-out expires.

'message-string' A character string unique to an I488 instrument, programming it to function in a specified manner (e.g., make a certain measurement, switch to a certain range, etc). Must be enclosed within single quote (') marks.

'command-string' A series of commands in the form of 3-digit octal numbers bracketed by % symbols. Must be enclosed within single quote (') marks. (I488 mnemonics cannot be used with SET CMD OUT=.) May also be the name of a variable containing the 3-digit octal code, e.g.,

```
SET CMD OUT=%CMDVAR%;
```

For definitions of the I488 command mnemonics, refer to para 6.3, I488. The I488 mnemonics and their octal values are:

Mnemonic	Octal Code	Description
GET	%010%	Group Execute Trigger.
DCL	%024%	Device Clear.
GTL	%001%	Go to Local.
MLA	%0YX%	My Listen Address.
MTA	%1ZX%	My Talk Address.
UNL	%077%	Unlisten.
UNT	%137%	Untalk.
LLO	%021%	Local Lock-Out.
MSA	%1YX%	My Secondary Address.
SDC	%004%	Selected Device Clear.

OXY, 1ZX, 1ZY from Table D-1.

Example

A short I488 programming sequence for the Fluke[1] 1953 counter appears in para 6.3, I488. The same example is given below with SET and MEAS used instead of READ and WRITE. Note that the SUBSTRing function must be used here to break the character string from the counter into the required character and numerical data. The program measures the frequency of a signal on the UUT and to print the results. Included is an explanation of the major functions performed by the program, followed by a listing of the test program statements.

1. Initializing the Program. The DECLARE statements are needed to define each variable used in the program. Note that the proper length should be DECLAREd for the character string variable used in storing received data from the device. If the DECLAREd variable specifies fewer characters than are actually received, all extra characters will be truncated from the string and lost. If the variable is DECLAREd longer than needed, the system will time-out (after 1 second) and whatever data have been received will be accepted as the complete character string.

[1]John Fluke Mfg. Co., Inc., Mountlake Terrace, WA.

2. Connecting the Instrument to the UUT. The functional output (or input) of each IEEE-488 device plugs into a BNC connector on the rear panel of the Processor Bay. As shown in Figure 6-3, these connectors form the inputs to the optional instrument multiplexer MUX(2). These inputs can then be connected to the appropriate MUX channels (A, B, C, or D) via a SET MUX statement and the MUX channels connected to the appropriate test pins via a SET SCAN statement. These 2 types of statements are included in the sample program for reference. The connections specified by these statements are not actually needed in this program, however, since subsequent statements place the instrument in self-test mode and no UUT measurements are made.

3. Handshaking between the systems and I488 Device. A series of CLEAR, SET, TEST and MEAS I488 statements directs the interaction between the I488 devices and the I488 controller in the test system. Briefly, for this typical example, the handshaking takes place as follows:

a. The CLEAR I488 statement activates the Interface Clear (IFC) line to all I488 instruments on the bus.

b. The next SET I488 CMD OUT statement initializes all devices via UNL, UNT commands and specifies device no.1 (MLA) as the device to which the next control message is being directed. In this example, the Fluke counter is assumed pre-wired for device address no.1.

c. The SET I488 OUT statement then transmits a control message to the addressed device (the Fluke counter), informing it of the type of measurement to be made. This message is in the form of a character string tailored to the particular device being addressed. The particular sequence of characters needed to perform a desired function must be obtained from the technical manual on that device.

In this example using the Fluke counter, the character string specifies the following functions:

F6	Self-test mode
R2	Range
A+1	Channel A setup information
LA+.5	Channel A voltage level information
B-1	Channel B setup information
LB+.5	Channel B voltage level information
S1M1H1	Operate mode
T	Trigger

d. The TEST I488 statement checks the I488 bus to determine if the Service Request (SRQ) line is active. If no request is present, the error branch transfers control to label NOSRQ. This increments a loop counter and causes the TEST I488 statement to be repeated up to 10 times before the SRQ line is considered to have failed. If, however, the device activates the request line, the next statement following TEST I488 is executed.

e. The next SET I488 CMD OUT statement changes the Fluke counter from listen to talk mode via an MTA code and the 227X to listen mode by MLA.

f. In response, the device places its measurement data on the I488 bus. The MEAS I488 INTO INFO statement then transfers the bus data into character string variable INFO, where it can be examined by the program.

2. All commands have a transaction timeout of 1 second. Any time the system waits more than 1 s for a command or a response to be issued and completed, a time-out will occur. If the time-out occurs when the controller is in talk mode, a system run-time error results. If the time-out occurs when the controller is in listen mode, the incomplete response is accepted as being complete and the program continues. The most common occurrence of the time-out condition results when a character string DECLAREd as n characters long receives fewer than n characters from the bus instrument. This can occur when an instrument drops leading zeroes from its measurement data.

3. When READ is used to convert a character string to a float (e.g., the READ ID=VALU'%F%'VAL statement in the counter example above, there must be no spaces between the sign character and the digits. If spaces are present, the programmer must eliminate them before the conversion. This can be done simply by extracting the sign from the character string and then forming a substring containing just the number. (Refer to the SUBSTR function in para 6.2.)

Table D-1
I488 DEVICE ADDRESS LIST

LISTEN ADDRESSES		TALK ADDRESSES		SECONDARY ADDRESSES	
Address	Code	Address	Code	Address	Code
0	%040%*	0	%100%*	0	%140%*
1	%041%	1	%101%	1	%141%
2	%042%	2	%102%	2	%142%
3	%043%	3	%103%	3	%143%
4	%044%	4	%104%	4	%144%
5	%045%	5	%105%	5	%145%
6	%046%	6	%106%	6	%146%
7	%047%	7	%107%	7	%147%
8	%050%	8	%110%	8	%150%
9	%051%	9	%111%	9	%151%
10	%052%	10	%112%	10	%152%
11	%053%	11	%113%	11	%153%
12	%054%	12	%114%	12	%154%
13	%055%	13	%115%	13	%155%
14	%056%	14	%116%	14	%156%
15	%057%	15	%117%	15	%157%
16	%060%	16	%120%	16	%160%
17	%061%	17	%121%	17	%161%
18	%062%	18	%122%	18	%162%
19	%063%	19	%123%	19	%163%
20	%064%	20	%124%	20	%164%
21	%065%	21	%125%	21	%165%
22	%066%	22	%126%	22	%166%
23	%067%	23	%127%	23	%167%
24	%070%	24	%130%	24	%170%
25	%071%	25	%131%	25	%171%
26	%072%	26	%132%	26	%172%
27	%073%	27	%133%	27	%173%
28	%074%	28	%134%	28	%174%
29	%075%	29	%135%	29	%175%
30	%076%	30	%136%	30	%176%

*Reserved for system addresses.

Production Test Division
300 Baker Avenue
Concord, Massachusetts 01742

USA

***Boston** • (Sales) Waltham, MA 02254
Tel: (617) 890-4900 • TWX: 710 324 0893

(Service) Concord, MA 01742
Tel: (617) 369-4400 • TWX: 710 347 1051

***Chicago** • 1156 Shure Drive, Arlington Heights, IL 60004
Tel: (312) 577-1881 • TWX: 910 291 1209

***Cincinnati** • Suite A, 2002 Ford Circle, Milford, OH 45150
Tel: (513) 831-9210 • TWX: 810 460 8323

***Colorado Springs** • 6180 Lehman Drive, Suite B207, Colorado Springs, CO 80907
Tel: (303) 593-8238

***Dallas** • 1121 Rockingham Lane, Suite 100, Richardson, TX 75080
Tel: (214) 234-3357 • TWX: 910 867 4771

***Los Angeles** • P.O. Box 19500, 17361 Armstrong Avenue,
Irvine Industrial Complex, Irvine, CA 92714
Tel: (714) 540-9830 • TWX: 910 595 1762

***New York** • 22-08 Route 208, Fair Lawn, NJ 07410
Tel: (NJ) (201) 797-8001; (NY) (212) 964-2722 • TWX: 710 988 2205

***Orlando** • 3751 Maguire Blvd., Suite 170, Orlando, Florida 32803
Tel: (305) 894-4303 • TWX: 810 850 0270

***San Francisco** • 2855 Bowers Avenue, Santa Clara, CA 95051
Tel: (408) 727-4400 • TWX: 910 338 0291

***Washington, D.C.** • 1701 Research Boulevard, Rockville, MD 20850
Tel: (301) 424-6224 • TWX: 710 828 9783

CANADA

***Toronto** • 307 Evans Avenue, Toronto, Ontario M8Z 1K2
Tel: (416) 252-3395 • TLX: 06-967624

Montreal • Tel: (514) 747-1052 • TLX: 05-826652

EUROPE

***England** • Norreys Drive, Maidenhead, Berkshire SL6 4BP England
Tel: (0628) 39181 • TLX: 851-848321

Best • Tel: (0) 49984240 • TLX: 844-59535

***Milan** • Tel: (02) 8466541 • TLX: 843-320373

***Munich** • Tel: (089) 41690 • TLX: 841-529917

***Paris** • Tel: (01) 7970739 • TLX: 842-220991

Rennes • Tel: (099) 535154

***Rome** • Tel: (06) 455839

Wiesbaden • Tel: (06121) 370057 • TLX: 841-04186489

***Zurich** • Tel: (01) 552420 • TLX: 845-53638

ASIA

Japan • Tokyo Electron Limited, Shinjuku-Nomura Bldg.,
1-26-2 Nishi-Shinjuku, Shinjuku-Ku, Tokyo 160, Japan
Tel: 343-4411 • TLX: 232-2240 Labtel J

Other countries in Asia and Pacific • Tel: 344-9245 (Tokyo)

LATIN AMERICA

Contact WALTHAM, MA 02254 USA
Cable GENRAD Waltham (MASS.)

*Service Facilities Available

Printed in USA